THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Mention : Informatique

École doctorale MATISSE

présentée par

Bartosz Grabiec

Préparée à l'Unité Mixte de Recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Supervision de systèmes répartis utilisant des dépliages avec contraintes de modèles temporisés

Thèse soutenue le 4 octobre 2011
devant le jury composé de :

Béatrice BÉRARD,
Professeur à l'UPMC Paris / *rapporteur*
Pierre-Yves SCHOBBENS
Professeur à l'Université de Namur / *rapporteur*

Stefan HAAR
Directeur de Recherche à l'INRIA de Saclay / *examinateur*
Axel LEGAY
Chargé de Recherche à l'INRIA de Rennes / *examinateur*
Didier LIME
Maître de conférences à l'Ecole Centrale de Nantes / *examinateur*
Stephan MERZ
Directeur de Recherche à l'INRIA de Nancy / *examinateur*

Claude JARD
Professeur à l'ENS Cachan - Bretagne / *directeur de thèse*

Contents

Acknowledgements

I would like to express my gratitude to every person who contributed to this dissertation.

In particular, I would like to thank my PhD advisor Claude Jard for his inspiration, friendly attitude, and warm support during all the period of my work on this thesis.

I would like to express my deep gratitude to Béatrice Bérard and Pierre-Yves Schobbens who did me the honor of being reviewers of this thesis. Equally, I would like to thank Stefan Haar, Axel Legay, Didier Lime, and Stephan Merz for accepting the participation in the jury of my thesis. At this point, I would like to thank Axel Legay, who supported me during writing this thesis.

I am also very thankful to all the members of IRISA who supported me during the preparation of the dissertation. I am particularly indebted to all staff in Distribcom.

Most of all, I want to express my deepest gratitude to my family and friends. I am very much obliged and infinitely grateful to my parents who were patiently waiting for me to finish the dissertation and provided great mental support.

Finally, the most important everyday support came from the most important person: Anne-Sophie. Thank you, Anne-Sophie, for tolerating me sitting at the computer all those months.

Présentation de la thèse

0.1 Systèmes répartis temps réel

De nos jours, il est indéniable que les systèmes répartis deviennent une partie intégrante de notre vie quotidienne. La société est de plus en plus dépendante de technologies complexes et cela ne concerne pas uniquement notre qualité de vie mais principalement notre sécurité. Dans notre environnement, il est aisé de trouver de nombreux exemples de tels systèmes, en commençant par les réseaux mobiles quasi omniprésents, pour finir par les systèmes de contrôle aérien. Malheureusement, comme l'ont démontré les études jusqu'ici, ce qui peut apparaître simple d'un point de vue utilisateur n'est pas si évident aux yeux du concepteur de tels systèmes complexes. La conception et l'implémentation de systèmes répartis temps réel représentent toujours un grand défi pour les concepteurs et les développeurs.

La variété d'applications pouvant être associées avec le domaine des systèmes répartis est si grande que leur classification précise est une tâche difficile. Néanmoins, il existe plusieurs divisions de base selon lesquelles les systèmes répartis temps réel sont souvent classifiés. Comme nous pourrons le voir, ces classifications résultent de divers aspects fonctionnels et non-fonctionnels des systèmes concernés.

La première division que nous mentionnons est la division liée au déploiement d'éléments composant l'architecture du système. Il existe fondamentalement deux types de systèmes répartis : centralisés et décentralisés.

Les systèmes centralisés sont largement basés sur l'architecture client-serveur. Dans ce cas, le serveur est un nœud qui fournit et réalise habituellement des services contractés par ses clients. Un système centralisé a une architecture hiérarchisée, au sommet de laquelle figure une unité principale de gestion. Contrairement aux systèmes centralisés, les systèmes décentralisés ne disposent pas d'une hiérarchisation claire avec un nœud parent qui se distingue. Les éléments d'un système décentralisé sont en général indépendants. Ils peuvent bien sûr communiquer entre eux. Mais de cette façon, ils prennent généralement des décisions indépendamment des autres composants du système. Un exemple de systèmes décentralisés récemment populaires est les réseaux *peer to peer* à travers lesquels les utilisateurs peuvent partager les fichiers. Habituellement, dans un tel réseau, chaque utilisateur ne dis-

pose que d'informations partielles concernant les fichiers disponibles dans le système. De plus, il n'y a pas de coordinateur unique et global.

Un autre critère de division de systèmes répartis fréquemment cité est la division au sein de systèmes homogènes et hétérogènes. Malgré les nombreux avantages des systèmes homogènes, nous devrons aborder les systèmes hétérogènes quand nous ferons référence à de grands systèmes physiquement répartis. Ces derniers incluent les réseaux mobiles, internet, les réseaux informatiques à grande échelle (par exemple GRID, récemment populaire ; voir [49]). La variété de logiciels et de matériel informatique au sein d'un tel système, qui est plus ou moins un réseau complexe de divers composants, entraîne de nombreuses difficultés. Une d'entre elles est que l'opération des services et des programmes coopérant entre eux peut varier de manière significative dans ses différentes parties. Dans les systèmes hétérogènes, il existe également d'autres problèmes qui ne s'appliquent souvent pas aux systèmes homogènes. La problématique de l'extensibilité est l'un d'entre eux. La gestion de tels systèmes est souvent difficile et nécessite l'aide de couche logiciel spécialisé.

Une des caractéristiques clés des systèmes répartis temps réel est également les contraintes temporelles affectées à certain états ou actions. Pour exemple, considérons un système en charge de la gestion d'un hôpital. Ces derniers temps, de tels systèmes sont composés de nombreux modules différents comprenant quasiment toutes les domaines de l'activité hospitalière : gérer les attentes des patients, maintenir le fonctionnement des cliniques, gérer les blocs opératoires, contrôler les machines dans les laboratoires, gérer le budget de l'hôpital, nourrir les patients, etc. De nombreuses contraintes temporelles s'appliquent à ces composants et processus fort divers. Certaines d'entre elles sont strictes ; et les dépasser peut mener à des conséquences désastreuses telles que le décès du patient. D'autres sont moins restrictives et les dépasser ne cause pas de graves problèmes. Afin de distinguer ces deux types de contraintes temporelles, nous utilisons respectivement le concept de systèmes temps réel strict et souple. En tant qu'exemple de systèmes temps réel strict, dans le cas où l'écoulement du temps joue un rôle crucial, nous pouvons citer les systèmes surveillant les fonctions vitales des patients et permettant une réponse relativement rapide des autres machines, ce qui améliore ainsi la condition du patient, ou encore appelant le personnel hospitalier. Nous pouvons citer le pacemaker en tant qu'exemple commun de système temps réel.

En tant qu'exemple élémentaire de systèmes temps réel souple, nous pouvons évoquer un système servant des repas au sein de l'hôpital. Dans cette configuration, le temps ne joue pas de rôle significatif. Ainsi, dans la majorité des cas, tout repas servi avec quelques minutes de retard ne génère pas de conséquences majeures. Un exemple similaire peut être le système de gestion de l'attente des patients. Dans ce cas, l'écoulement de certaines contraintes temporelles planifiées dans de tels systèmes ne cause pas non plus

de conséquences significatives.

Comme mentionné, la modélisation de systèmes répartis temps réel nécessite souvent l'utilisation de diverses contraintes temporelles. Dans une des parties du document (voir Chapitre 2), nous pourrons remarquer que divers types de contraintes temporelles ont été introduits, ainsi que de nombreux types de modèles formels. Ces restrictions diffèrent dans les expressions utilisées pour les formuler. Elles varient de simples expressions linéaires à, par exemple, des expressions plus complexes avec dérivées. Il a été montré de nombreuses fois que la complexité de telles expressions affecte directement les possibilités d'analyse et de vérification de modèles les utilisant.

Il faut noter que les contraintes temporelles sont en résumé les fonctions opérant sur les valeurs des horloges physiques disponibles dans le système. Cependant, à ce niveau peuvent surgir certaines difficultés. Étant donné que le système est réparti, il peut avoir de multiples horloges indépendantes. Imaginons une situation dans laquelle le système réparti soit composé de plusieurs ordinateurs. Chaque ordinateur dispose de sa propre horloge locale pouvant être lue ou modifiée. Lorsqu'on modélise alors le système ainsi que ses contraintes temporelles, il est possible qu'un programme sur un ordinateur veuille lire la valeur de l'horloge présente sur un autre ordinateur. Or, comme il est connu dans le domaine des protocoles de synchronisation d'horloge, ceci n'est pas tâche facile. Cependant, en pratique, l'opération faite à distance de lecture directe de la valeur de l'horloge n'est pas si standard. En général, ceci est dû au fait qu'il existe une couche supplémentaire logicielle ou de matériel informatique responsable de la synchronisation de toutes les horloges. De cette façon, tous les ordinateurs ont accès à l'horloge virtuelle qui fournit un point de référence commun. Bien sûr, une telle solution n'est pas parfaite. En effet, la synchronisation de l'horloge est parsemée d'erreurs résultant notamment des délais de communication entre les ordinateurs, des différences entre les paramètres des horloges physiques (qui influencent par exemple la fréquence des horloges), voire des différences entre les paramètres de l'environnement dans lequel sont situés les ordinateurs (par exemple la température ambiante).

Les questions liées aux protocoles de synchronisation temporelle dans les réseaux informatiques ont été assez bien étudiées. Parmi les protocoles de synchronisation les plus populaires, se dénote de façon particulièrement remarquable le protocole NTP[1]. Ce dernier, décrit dans [74], est devenu un standard largement répandu. Les solutions utilisées dans NTP témoignent de la complexité du problème de synchronisation d'horloges dans un environnement réparti.

Malheureusement, seules quelques incertitudes associées à la synchronisation d'horloges restent solvables grâce aux protocoles de synchronisation d'horloge. Étant donné que, dans la plupart des cas, l'incertitude concernant

[1]Network Time Protocol

la mesure du temps est inévitable dans les systèmes répartis, il est naturel que le problème affecte également les modèles formels utilisés pour les représenter. Dans ce but, un nombre de modèles formels a été introduit. Ces derniers s'attaquent directement à certains aspects concernant la fiabilité des mesures du temps dans les environnements répartis.

Une caractéristique importante des systèmes répartis est leur imprévisibilité, leur non-déterminisme. Par exemple, considérons la communication entre des composants d'un système réparti. Celle-ci est généralement asynchrone et il est parfois impossible d'estimer de façon précise le délai au bout duquel les données sont reçues par son destinataire, en supposant que ces données sont effectivement reçues. Des pannes et des retards n'ont rien d'inhabituel dans de systèmes aussi grands.

Dans le cas de systèmes plus petits tels que les multiprocesseurs avec mémoire partagée, la communication est souvent synchrone. Dans ce cas, il est plus aisé de planifier certaines procédures et de prévoir la durée de leurs opérations.

L'architecture moderne des systèmes répartis est largement basée sur un réseau d'ordinateurs interconnectés pouvant échanger des données relativement librement. Chaque ordinateur exécute un certain nombre de programmes. En général, chacun de ces programmes consiste en un ou plusieurs processus pouvant communiquer entre eux. De plus, il est possible de trouver fréquemment au sein des processus des *threads* fonctionnant simultanément.

Comme nous pouvons le voir, la communication et la simultanéité peuvent se produire au sein de systèmes répartis à différents niveaux d'architecture. D'une part, il s'agit d'une conséquence de multiples utilisateurs de tels systèmes. D'autre part, les utilisateurs peuvent simultanément initier de nombreuses tâches. Un facteur important est la dispersion géographique des nœuds intervenant dans le processus. Il s'avère que ce facteur est relativement important, notamment dans le contexte des contraintes temporelles imposées au système réparti. Ceci est largement dû au temps de communication qui affecte fortement la vitesse globale du traitement de l'information.

0.2 Supervision

Lors de la conception de systèmes répartis temps réel, il est difficile de prévoir toutes les situations pouvant causer un dysfonctionnement de tels systèmes. Le fait qu'un système soit réparti entrave déjà significativement le travail de ses concepteurs et testeurs. De plus, si nous mentionnons les contraintes temporelles et le fait que l'environnement dans lequel le système fonctionne peut subir des changements constants, l'analyse statique précise d'un tel système semble quasiment irréalisable. Souvent, il est impossible d'analyser dans un tel système un problème aussi simple que l'atteignabilité des états. À ce niveau, la seule vérification possible du système est sa surveillance durant son

fonctionnement, généralement au sein d'une portée limitée qu'il est possible de plus ou moins prévoir (par exemple, dans un délai imparti et avec des valeurs restreintes de certains paramètres). C'est l'une des raisons pour lesquelles nous traiterons dans une partie du document d'aspects sélectionnés de la supervision de systèmes répartis.

Avant de décrire certains détails rattachés à la supervision de systèmes répartis temps réel, nous introduisons plusieurs termes utilisés dans notre travail. Un système supervisé est un système qui est surveillé afin de détecter un ensemble de comportements pouvant par exemple causer un dysfonctionnement. Un système de supervision (ou de surveillance) est l'ensemble des notions et des éléments servant à surveiller un système supervisé, et qui permet d'analyser les observations collectées dans le but d'identifier certains événements et comportements.

Afin de surveiller et diagnostiquer efficacement un système, le système de surveillance doit disposer d'informations adéquates concernant les événements se produisant. La décision concernant le type d'information collectée et analysée par le système de supervision appartient généralement au concepteur du système. Bien sûr, il est important que l'information soit utile dans la détection d'une cible potentielle habituellement fausse et d'opérations involontaires du système. Il faut aussi noter que rassembler un nombre excessif d'informations dans le but de reproduire les actions du système s'avère dans de nombreux cas inutile, notamment quand il s'agit de systèmes temps réel. Cela découle du fait que l'opération de tels systèmes est généralement non entièrement déterministe. Et il est fréquemment impossible de recréer les conditions dans lesquelles le système a fonctionné.

Comment est l'information concernant les événements contenus dans l'observation réalisée ? Il existe fondamentalement deux sources d'origine des événements. La première est basée sur les mécanismes, les procédures qui sont directement intégrés dans le système supervisé. Par exemple, nous pouvons considérer un programme à l'intérieur duquel le programmeur a intégré des procédures spéciales fournissant des informations concernant les événements. La seconde source d'événements est basée sur les mécanismes externes. Typiquement, il s'agit de solutions basées sur le matériel informatique et n'interférant pas avec l'opération du système surveillé. Cependant, cette solution est habituellement moins flexible et plus onéreuse.

Comment un système de surveillance collecte-t-il et traite-t-il l'information concernant les événements ? Un système réparti typique est composé de nombreux dispositifs fonctionnant indépendamment et échangeant parfois leurs données. Les observations réalisées par ces appareils à l'intérieur de la procédure de supervision peuvent être exploitées de différentes manières.

Nous pouvons distinguer deux scénarios de base : elles peuvent par exemple être stockées localement sous la forme de traces locales et attendre un traitement ultérieur ; elles peuvent aussi être directement envoyées vers une unité centrale de surveillance où elles peuvent être analysées toutes ensemble.

Un avantage concernant le premier cas est qu'un coût de communication inutile peut parfois être évité si certaines propriétés recherchées peuvent être détectées localement sans se référer à l'ensemble du système. Cependant, à un certain point, il peut s'avérer nécessaire d'envoyer des résultats à une autre unité de surveillance afin d'obtenir une vision globale de certains événements se produisant. Malheureusement, ceci peut facilement mener à un transfert soudain d'un très grand nombre de données.

Dans le second cas, les observations sont directement transmises à un superviseur sans avoir reçu de traitement local au préalable. De plus, cette approche comporte le risque de surcharger le système avec un grand nombre d'observations transmises à un système de surveillance. Or, contrairement à la première approche, la charge du système, notamment les liens de communication, peut être significative durant toute la période où le système est actif. Par conséquent, une solution raisonnable semble être une répartition partielle de surveillance, ce qui permet ainsi d'éviter les inconvénients des systèmes centralisés. Cependant, il faut noter qu'une telle configuration n'est pas toujours possible, notamment de par l'architecture du système ou encore de par une spécificité de la surveillance.

Dans notre travail, nous nous focaliserons sur le cas spécifique de la supervision basée sur des modèles. Le schéma général d'un système de surveillance basée sur des modèles est présenté dans la Figure 1. Comme nous pouvons le constater, il existe quatre éléments principaux dans un tel système de surveillance :

- *Le modèle du système supervisé.* En d'autres termes, il s'agit d'une représentation formelle du système de surveillance qui peut être décrite par exemple par les réseaux d'automates temporisés ou bien les réseaux de Petri temporels (pour plus de détails, voir Chapitre 2) ;

- *Les observations*, à savoir les informations concernant les événements produits par des composants du système ;

- *Les explications* — ensemble de scénarios produit sur la base du modèle du système et des observations. Généralement, une explication est une collection d'événements comprenant une relation d'ordre entre ces derniers. Étant donné que nous étudions les modèles temporisés, il existe également des contraintes temporelles associées à des explications. Les contraintes temporelles permettent d'assigner un horodatage à des événements spécifiques dans les explications. Il convient de remarquer qu'il peut y avoir diverses explications pour un seul ensemble d'observations. En effet, de nombreuses trajectoires d'un modèle donné peuvent produire le même ensemble d'observations ;

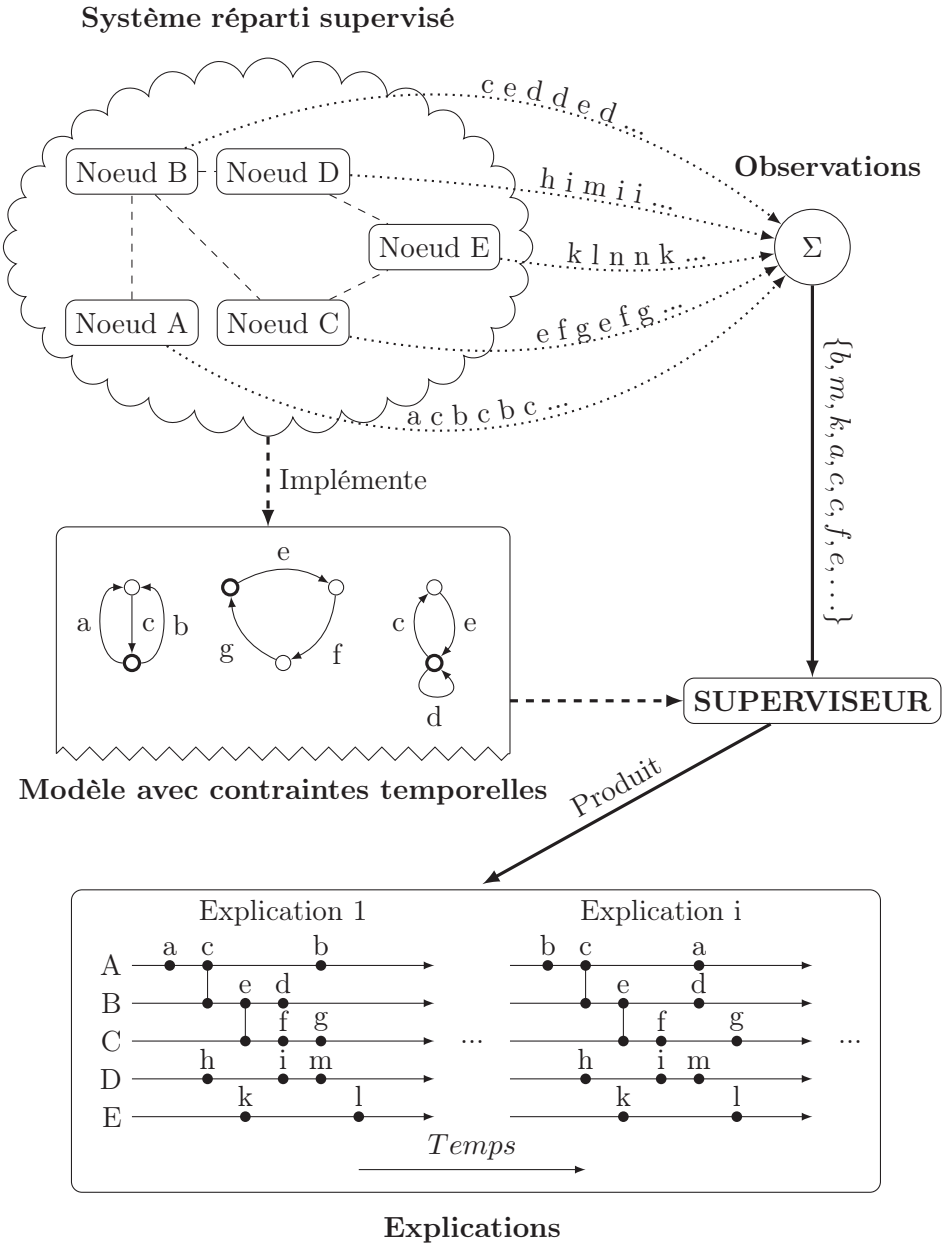


Figure 1: Supervision d'un système réparti basée sur un modèle

- *Le module de surveillance* : en utilisant les connaissances sur le modèle du système, il analyse les observations dans le but d'obtenir des explications concernant le comportement de ce système.

La surveillance de systèmes répartis temps réel implique de nombreuses problématiques. Comme mentionné plus haut, la surveillance est faite lorsque le système supervisé fonctionne. Cela permet à ce système d'interférer avec le fonctionnement du système de surveillance. Comme nous pouvons le supposer, ceci cause en contre-partie une distorsion des résultats de la surveillance, selon le principe bien connu d'incertitude. Cela provient du fait que la production d'informations concernant les événements ensuite analysés par le système de surveillance nécessite un certain laps de temps. Malheureusement, les opérations se déroulant au sein du système sont ralenties par le système de surveillance alors que le temps s'écoule. En revanche, cette situation peut considérablement affecter les résultats du système de surveillance. La seconde raison pour laquelle le système de surveillance peut perturber le fonctionnement du système supervisé concerne le processus de recueil d'informations concernant les événements du système. Autrement dit, une grande quantité d'informations décrivant les événements se déroulant au sein du système, ainsi que l'utilisation des mêmes liens de transmission pour le système de surveillance et le système supervisé, impliquent que le temps nécessaire pour transmettre les données utilisées par ledit système supervisé peut être considérablement plus long en comparaison avec la situation où il n'y a pas de système de surveillance.

Comme mentionné précédemment, un facteur entravant un système réparti de surveillance temps réel est le manque d'horloge globale avec laquelle un individu pourrait mesurer temporellement la survenue d'un événement.

Comme le démontrent de nombreuses expériences de concepteurs et de programmeurs, le fait même que le système soit basé sur des restrictions relatives au temps réel est un défi majeur concernant les systèmes de surveillance. En ajoutant à cela que nous traitons d'un système réparti, la procédure entière de surveillance devient considérablement compliquée. Si nous étudions les procédures permettant le débogage de programmes ou de programmes distribués, nous découvrons qu'elles consomment relativement beaucoup de temps, et qu'elles peuvent retarder significativement le travail réalisé par le programme alors debogué.

0.3 Choix des modèles

Dans la section précédente, nous avons décrit les nombreux éléments composant le système de surveillance dont nous traitons. Une des principales hypothèses d'un tel système est l'existence d'un modèle du système supervisé. Cependant, cela soulève la question du type de formalisme devant être utilisé pour modéliser le système de façon à ce que toutes les exigences du

système de surveillance soient satisfaites. La littérature à ce sujet fournit différentes possibilités. Parmi elles, nous pouvons trouver deux modèles récents très connus et fréquemment utilisés : les réseaux d'automates temporisés et les réseaux de Petri. Le choix d'un de ces modèles dépend de nombreux facteurs tels que les caractéristiques du système supervisé ou les propriétés qui sont surveillées.

Les réseaux d'automates temporisés et les réseaux de Petri peuvent modéliser les systèmes répartis avec contraintes temporelles. Dans ce cas, un aspect important repose certainement sur les possibilité de modéliser les événements parallèles.

Par exemple, imaginons qu'il y ait un système réparti dans lequel figurent de nombreux processus parallèles. De temps à autre, ces processus interagissent entre eux. Nous supposons que chacun de ces processus se compose d'une séquence d'actions. Dans cette situation, il semble presque naturel de modéliser chaque processus en tant qu'automate. De plus, en cas d'interaction entre automates, une simple synchronisation d'actions peut être utilisée.

Afin de modéliser convenablement le système réparti temps réel, l'aspect temporel doit être pris en compte. Les deux types de modèles cités précédemment fournissent la possibilité d'imposer les contraintes temporelles sur les événements et les états qu'ils génèrent. Cependant, il faut noter que, pour chacun des modèles, ces restrictions sont différentes (plus de détails dans le chapitre suivant).

Dans la section précédente, nous avons brièvement mentionné le rôle joué par le processus d'acquisition de l'observation dans le système de surveillance. Nous avons identifié les modalités possibles d'observation. Néanmoins, une question subsiste à savoir quelle forme une information concernant les événements se produisant dans le système devrait avoir de sorte qu'elle soit utile pour des analyses ultérieures. Comme nous le savons, dans les systèmes répartis, il est difficile d'établir un point de référence commun pour tous les événements tels que le temps par exemple. Y compris dans les situations où le système de surveillance dispose d'informations concernant le temps de survenue de certains événements, ces derniers ne sont habituellement pas assez détaillés pour identifier les possibles erreurs dans l'opération du système de surveillance. Heureusement, sans même l'aide d'horloges physiques, le système de surveillance peut obtenir une autre information importante concernant les événements. En effet, en utilisant par exemple des horloges vecteurs ([48, 69]), nous pouvons explorer les dépendances causales entre les événements. De cette manière, deux types élémentaires de relations entre événements peuvent être distingués :

- la relation $a \leq b$ signifie que l'événement a précède de façon causale l'événement b ;
- la relation $a \mathsf{co} b$ signifie que les deux événements a et b sont exécutés en parallèles.

Il peut également arriver que la relation causale entre les événements ne soit pas connue. Dans ce but, nous utilisons le caractère ?. En d'autres termes, l'expression $a?b$ signifie que la relation causale entre a et b est inconnue. Comme nous le verrons, les relations présentées jouent un rôle fondamental dans la description des problèmes associés aux systèmes répartis. Les dépendances entre événements que nous avons introduites définissent ladite relation d'ordre partiel. Ainsi, la relation permet de décrire l'ordre des événements dans un système réparti et, à ce titre, elle est l'un des points centraux dans les structures utilisées pour stocker et traiter les informations concernant les événements.

En utilisant les relations présentées ci-dessus dans le cas de l'observation, il faut noter que, dans le meilleur des cas, uniquement sur la base d'observations, nous connaissons les dépendances causales exactes entre tous les événements. Et, dans le pire des cas, l'ordre des événements est totalement inconnu. Pour connaître l'ordre des événements, nous pouvons utiliser dans ce but un système de surveillance tentant de calculer les dépendances recherchées sur la base du modèle du système et des observations.

Dans la suite du document, nous supposerons en général que la relation causale entre les événements présents dans les observations est inconnue. Naturellement, dans de nombreuses situations, cela implique un plus grand nombre d'explications possibles compatibles avec une observation donnée. D'autre part, cette approche montre les vastes capacités du système de supervision, à la fois concernant l'aspect théorique et la possibilité d'implémentation.

Dans le paragraphe précédent, nous avons brièvement mentionné le concept de dépliage qui est une structure rassemblant les événements se déroulant dans le système et les relations causales entre ces événements. Il est caractéristique de cette structure que chaque événement y ait une histoire unique. En d'autres termes, pour un événement donné provenant d'une telle structure, nous pouvons clairement tracer le processus par lequel est survenu l'événement. Étant donné que nous introduisons dans ce document le problème de surveillance dans le contexte des deux modèles, nous introduisons également deux méthodes similaires permettant de formaliser les explications.

Afin de représenter les explications produites par le système basé sur les réseaux de Petri, nous utilisons lesdits réseaux d'occurrence. Il s'agit d'une structure représentant les survenues d'événements dans le système et les états les accompagnant. Comme mentionné plus haut, étant donné que chaque événement a son propre passé lorsque le modèle contient des contraintes temporelles, la situation devient plus complexe. Considérons une situation simple dans laquelle nous traitons de l'action pouvant se produire durant une période de temps donnée. Si nous voulions décrire toutes les survenues possibles de l'action en utilisant un simple réseau d'occurrence, le nombre d'événements correspondants pourrait être infini dans le pire des cas. À cet effet, nous utiliserons la variante symbolique des réseaux d'occurrence dont

la structure regroupe les événements avec un temps d'exécution compris dans un certain intervalle de temps. Ainsi, au lieu d'énumérer tous les événements pouvant se produire durant une période de temps spécifique, nous pouvons définir un événement symbolique avec une contrainte temporelle symbolique représentant tous les temps de réalisation possibles de l'événement considéré.

Nous disposons d'une situation similaire dans le cas de réseaux d'automates temporisés. Cependant, comme nous le verrons dans un prochain chapitre, la structure utilisée pour représenter les événements et les états des réseaux d'automates temporisés est légèrement différente. Pour expliquer rapidement cette différence, au sein des réseaux d'occurrence, chaque condition ou événement représente un élément unique du modèle associé qui, dans notre cas, est un réseau de Petri. D'un autre côté, dans le cas d'une structure d'événements utilisée pour les réseaux d'automates, chaque élément d'une telle structure peut représenter à la fois plusieurs éléments du modèle associé. Chaque élément peut regrouper des transitions et des places d'automates.

Comme nous le savons concernant le résultat du processus de surveillance, nous obtenons un ensemble de scénarios. À leur tour, ces scénarios incluent des événements reflétant le comportement possible du système, par exemple les explications. Naturellement, les explications doivent être cohérentes avec les observations émanant du système supervisé. Dans ce but, il ne s'agira alors que d'un dépliage incluant des événements cohérents avec l'observation. Cette structure est basée sur la notion de dépliage, avec pour différence de prendre en compte l'observation.

Un exemple d'introduction

L'exemple que nous présentons ci-après est inspiré d'un problème réel. Néanmoins, nous le présentons sous forme d'une version très simplifiée dans le seul but de présenter l'idée des problèmes que nous traitons dans le suite de notre travail. Nous montrons comment le concept de réseau d'automates temporisés peut être utilisé dans un processus simple de production. Ce type de problème peut être considéré comme une simple variante d'un problème d'ordonnancement dans lequel existent des ressources devant être gérées de façon appropriée. Le système que nous considérons est représenté dans la Figure 2 et se compose de trois types basiques de composants : un fournisseur, un cahier des charges et des machines. Chaque composant contient plusieurs automates qui, tous ensemble, forment un réseaux d'automates. Chaque cercle représente un état et chaque flèche représente une transition entre états. Toutes les flèches sans état d'entrée désignent les états initiaux. Les flèches en caractère gras représentent les transitions spéciales appelées synchronisations. De plus, les transitions concernant une synchronisation ont

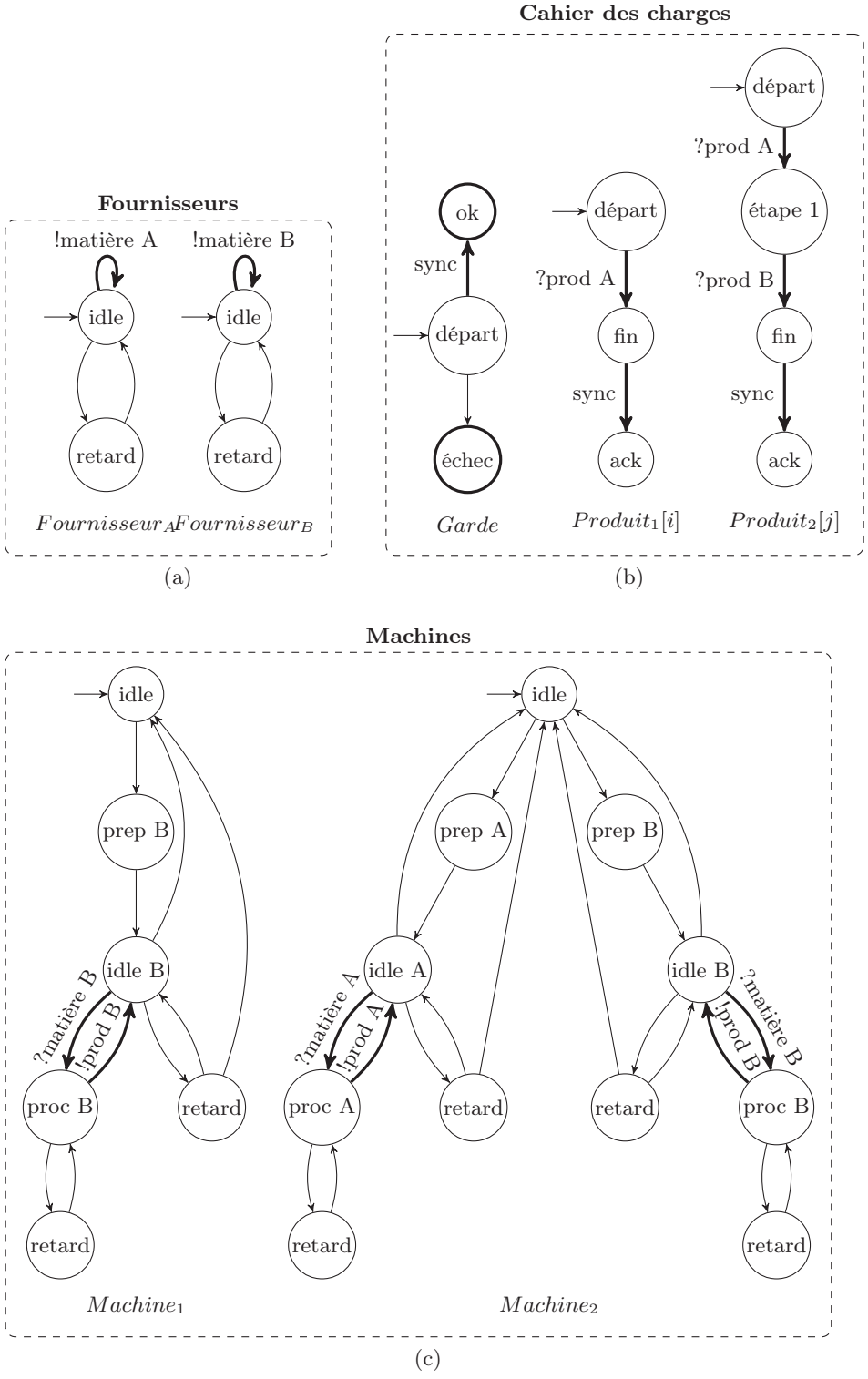


FIGURE 2 – Un système simple de production

la même étiquette. Dans notre exemple, nous avons trois types de synchronisation : *matière X*, *prod X*, *sync*, où X est égal à A ou B. Dans le cas de notre exemple, nous avons ajouté deux caractères avant certains noms de transitions, à savoir " ?" ou " !". Dans le cas de synchronisations contenant des transitions avec ces caractères, nous supposons que la synchronisation peut impliquer uniquement deux transitions avec le même nom, l'une avec " !" et l'autre avec " ?" sur l'étiquette. D'autre part, la synchronisation sync représente l'exécution de toutes les transitions avec ce nom.

Nous décrivons brièvement ci-dessous tous les composants :

- *Les fournisseurs* — Leur tâche est de fournir des matières premières qui sont ensuite traitées par des machines en produits finis. Nous distinguons divers types de matières. Dans notre exemple, nous utilisons deux types d'entre eux, à savoir : *matière A*, *matière B*. Selon le type de matière, cette dernière peut être traitée par différents types de machines. Il arrive parfois que les fournisseurs aient des difficultés à fournir certaines matières, causant alors des retards dans la production. De telles situations peuvent naturellement avoir un impact sur le cahier des charges de production.

- *Le cahier des charges* — Il montre combien de détails de chaque type doivent être produits dans une période de temps donnée. En d'autres termes, il représente les échéances devant être respectées. Dans la Figure 2b, nous pouvons observer trois types d'automates. Deux d'entre eux représentent un processus de production de deux sortes de produits finis. Le premier produit $Produit_1$ ne nécessite que d'un demi-produit *prod A* pour être terminé, tandis que $Produit_2$ nécessite un demi-produit *prod A* puis un produit *prod B*. Nous supposons qu'il existe un certain nombre de chaque type de produits à être réalisé. Quand tous les produits finis sont terminés (voir les états avec les étiquettes *fin* dans la Figure 2b), il reste la dernière opération nommée *sync* utilisée afin de confirmer que le cahier des charges tout entier a été respecté à temps. L'automate utilisé en tant que chronomètre est nommé *Garde* et comporte deux transitions : *ok* si l'échéance est respectée, et *échec* le cas échéant.

- *Les machines* — Elles sont utilisées pour traiter les matières. Chaque machine peut seulement travailler certains types de matières. Dans notre exemple, $Machine_1$ accepte uniquement les matières de type B, tandis que $Machine_2$ accepte les deux types de matière, à savoir A et B. Avant qu'une machine ne débute la production, un certain laps de temps est nécessaire afin de la préparer (états étiquetés avec *prep X*). Ensuite, elle peut traiter une par une toutes les matières délivrées et les transformer en demi-produits. Cette procédure est représentée sous la forme d'une boucle simple composée de deux transitions : *matière X* et

prod X. Dans notre exemple, nous considérons également de possibles pannes des machines qui peuvent alors causer des retards significatifs (états étiquetés avec *retard*). Le point important est que nous supposons que les retards ne sont pas toujours observables et peuvent être difficiles à détecter durant la procédure de production.

Le système présenté dans l'exemple est statique dans le sens où le nombre de composants ne change pas durant son activité. Dans la description des composants, nous avons mentionné des contraintes temporelles rattachées à certaines transitions et états. Détenir l'information concernant les contraintes temporelles présentes dans le modèle est très important étant donné qu'elles rendent le modèle plus réaliste et reflètent finalement les possibles problèmes avec plus de précision.

Le système que nous présentons est relativement simple. Cependant, et ceci y compris dans l'exemple d'un tel système, nous pouvons déjà analyser quelques problèmes pouvant survenir dans le contexte de systèmes de surveillance avec contraintes temporelles. Par exemple, nous avons mentionné plus haut qu'il pouvait exister différents types d'échecs dans le système pouvant causer des retards. L'utilisateur du système peut par exemple se poser les questions suivantes : en cas de panne d'une machine, combien de temps cela prend-t-il pour la réparer de sorte que le cahier des charges soit respecté à temps et que la production n'ait pas à être reportée sur d'autres machines ; si le temps attribué pour la réparation est écoulé, reporter la production sur une autre machine est-il suffisant pour satisfaire les contraintes du cahier des charges ; si des retards ont été constatés, à quel moment de la procédure de production ces derniers sont survenus.

0.4 Positionnement et contribution

Dans notre travail, nous avons décidé de baser notre solution sur la théorie des dépliages que nous avons estimée la plus adaptée pour l'objectif que nous souhaitons atteindre. Cette théorie a été initialement introduite par McMillan dans [70], puis ensuite étendue et améliorée, notamment par Esparza dans [42]. À l'origine, les dépliages étaient utilisés pour la vérification de modèles en tant qu'alternative, plus spécifiquement pour les méthodes basées sur une sémantique séquentielle. Les dépliages se sont avérés plus concis que d'autres structures quand il s'agit de besoins de mémoire. Ils nous permettent également de stocker des informations sous la forme d'une structure d'événements partiellement ordonnée, alors que la structure basée elle-même sur des structures entièrement séquentielles ne peut stocker d'informations concernant les événements parallèles. Un des principaux recueils proposant de synthétiser les principales connaissances concernant les dépliages a été publié par Esparza et Heljanko ([44]).

La théorie des dépliages a trouvé quelques applications peu de temps après son introduction. L'application qui est particulièrement dans notre champ d'intérêt est la diagnosticabilité de systèmes répartis décrite par [15]. La supervision de systèmes répartis temps réel que nous présentons plus tard dans le document est une continuation de notre travail. La notion de diagnostic de systèmes répartis a été décrite dans [46]. Cependant, ce travail ne prend pas en compte les contraintes temporelles qui jouent souvent un rôle clé dans la surveillance de systèmes répartis temps réel.

Par exemple, le problème des dépliages et des contraintes temporelles dans le contexte des réseaux de Petri a été étudié dans [34, 32]. Dans ces travaux, une nouvelle technique de dépliage pour les réseaux de Petri et un exemple d'application à buts diagnostiques ont été proposés. Cette structure diffère des dépliages typiques des réseaux de Petri de par le fait qu'elle contient des informations concernant les contraintes temporelles. Malheureusement, quand nous considérons un dépliage de réseau de Petri temporel et que nous en retirons toutes les contraintes temporelles, il se peut que la structure restante soit substantiellement plus grande que le dépliage construit sur la base du même réseau de Petri sans contraintes temporelles. Ceci est dû à la duplication de certains événements dans le but de leur associer les contraintes temporelles adaptées. Le problème des duplications a été résolu, notamment dans [85, 84]. Cette nouvelle **méthode de construction de dépliages de réseaux de Petri temporels**, proposée dans ces travaux, résout le problème d'événements multiples au détriment de contraintes temporelles plus complexes. Comme nous l'aborderons plus loin dans le document, les dépliages construits de cette manière ont un grand avantage sur la première approche. En effet, si nous soustrayons les contraintes temporelles de la structure, il s'avère que sa taille n'est pas plus grande que celle du dépliage de la version non temporisée du modèle. Cela nous permet de diviser si nécessaire la construction de dépliages en un certain nombre d'étapes. Tout d'abord, nous construisons le dépliage d'une version non temporisée sous-jacente du modèle. Nous ajoutons ensuite les contraintes temporelles appropriées. Il faut noter qu'aucun nouvel événement n'est ajouté à la structure après la première étape.

Une des principales parties de notre travail est l'étude des **dépliages des réseaux d'automates temporisés utilisés pour réaliser la supervision**. Pour la première fois, le concept de tels dépliages a été proposé dans [29, 26]. Nous avons décidé d'étendre ce travail et de l'adapter au problème de surveillance de systèmes répartis. Dans [52], ce travail nous a menés à une technique de construction de dépliages guidés par une observation et une méthode permettant d'inférer les possibles dates de survenue d'actions, ceci en utilisant des contraintes symboliques dans les réseaux d'automates temporisés. Ceci a également été le début de notre travail ultérieur sur les **dépliages avec contraintes** qui est le nom attribué à ce type de dépliages. Ce travail a abouti à un ensemble d'algorithmes dédiés aux dépliages de

réseaux d'automates temporisés, ainsi qu'à leur implémentation.

Dans le Chapitre 4, nous abordons le problème des événements inobservables, c'est-à-dire les événements qui sont reflétés et visibles dans un modèle donné, mais qui ne peuvent être observés durant l'opération du système associé. Le problème d'observation partielle est déjà apparu de nombreuses fois dans la littérature et a été discuté dans le contexte de divers modèles formels, comme par exemple dans le domaine des réseaux de Petri ([37]) et des automates ([19, 27, 41, 18]). Cette problématique a également été abordée notamment dans les travaux sur la vérification de modèle ([43]), ou la supervision avec contrôle ([28]). Cependant, une grande partie de ces événements invisibles est ignorée et il n'existe pas d'information directe les concernant. Un exemple d'une telle approche est la supervision avec observateurs décrite dans [28]. Intuitivement, dans le but de créer l'observateur, un modèle de système est considéré et toutes les transitions correspondant à des événements inobservables sont alors retirées. Par conséquent, les états entre lesquels figurent des transitions inobservables sont regroupés. Ceci crée de nouveaux états regroupant des états d'origine. Il faut noter que, dans le cas d'un observateur ainsi défini, il n'existe aucune information directe concernant les événements pouvant se produire dans le système, mais pouvant être observés. De plus, dans ce cas, un inconvénient supplémentaire est la perte de certaines informations concernant les relations de causalité entre les événements. Par exemple, il est impossible de déterminer si certains événements ont été exécutés en parallèle.

Une autre approche du problème des événements inobservables a également été proposée par Wimmel dans [88]. Dans son travail, il propose d'ôter les transitions invisibles de sorte que le modèle modifié soit équivalent à un *pomset*. Intuitivement parlant, les relations causales sont conservées entre les événements. Cependant, le cas étudié par Wimmel est restreint à un cas spécifique de réseaux de Petri. Pour autant que nous le sachions concernant le cas général des réseaux de Petri, il semble que le problème n'ait toujours pas de solution.

Pour conclure, les informations concernant les événements inobservables peuvent être cruciales pour détecter quelque opération anormale du système telle que les retards cachés pouvant souvent se glisser dans le système et dont il est impossible de suivre la trace par le système de surveillance. Ainsi, retirer de tels événements peut ne pas être souhaitable. Cependant, un problème subsiste étant donné que certains événements inobservables dans le système peuvent être répétés et théoriquement produire des boucles inobservables infinies. Pour cette raison, nous proposons dans le Chapitre 4 une nouvelle approche qui consiste à **déplier un réseau de Petri guidé par une observation partielle**. Ce type nommé **dépliage partiel** nous permet de stocker et de traiter les informations concernant des boucles infinies et inobservables en utilisant une quantité finie de mémoire. Dans ce chapitre, nous abordons plusieurs classes de réseaux de Petri en montrant que

la complexité du modèle influence la complexité du problème. Cette partie du document n'a pas encore été publiée.

Dans la dernière partie de notre travail, à savoir le Chapitre 5, nous nous focalisons sur l'**implémentation des solutions proposées**. Jusqu'ici, relativement peu d'outils universels ont été créés, basés sur la théorie des dépliages, et qu'il serait facile de comparer en terme d'utilisabilité et de performance. Parmi les outils existants, nous pouvons par exemple trouver l'outil PEP (Programming Environment based on Petri Net ; voir [83, 54]), récemment Roméo (outil pour l'analyse de réseaux de Petri temporels ; voir [50, 2]) que nous mentionnons également plus loin dans notre travail, le Model-Cheking Kit qui est une collection de programmes incluant des algorithmes de vérification basés sur la théorie des dépliages ([81]). Parmi les diverses applications de dépliages, nous pouvons citer le travail susmentionné sur le diagnostic des systèmes à événements discrets ([15]), les travaux sur les dépliages utilisés dans le contexte des circuits logiques asynchrones ([61, 62]), ainsi que la vérification de systèmes mobiles ([73]).

Durant notre recherche, nous avons implémenté et vérifié des algorithmes **dépliant des réseaux d'automates temporisés en présence d'observations**. Avec la coopération de l'Institut de Recherche en Communication et Cybernétique de Nantes qui est l'auteur de l'outil Roméo, nous avons réalisé des tests sur la **construction de dépliages avec contraintes pour des réseaux de Petri temporels paramétrisés**. Nous avons également créé un outil expérimental que nous avons utilisé pour calculer des **dépliages avec contraintes de réseaux de Petri avec observation partielle** décrits dans le Chapitre 4.

De plus, nous avons décidé de restructurer et d'améliorer l'outil permettant de déplier les réseaux d'automates temporisés. De cette façon, l'**outil** Spinta a été créé. Ce dernier peut opérer sur des réseaux d'automates avec des paramètres et des systèmes de contraintes linéaires, c'est-à-dire les polyèdres.

Tous les outils mentionnés précédemment ont été utilisés, notamment pour préparer les études de cas et les exemples présentés dans ce rapport.

0.5 Organisation du document

Le Chapitre 2 commence par une brève présentation de notions de base que nous utilisons dans les chapitres suivants. La Section 2.1 est une courte introduction aux systèmes de transition temporisés qui sont un point de référence pour une discussion ultérieure concernant d'autres modèles. Ensuite, les Sections 2.2 et 2.3 présentent deux familles différentes de modèles connus destinés à la modélisation de systèmes répartis temps réel, à savoir les réseaux d'automates temporisés et les réseaux de Petri temporels. Suite à l'introduction de ces modèles, nous décrivons brièvement dans la Section

2.4 des résultats de base concernant la décidabilité de divers problèmes dans le contexte des deux modèles. Nous présentons ensuite très rapidement certaines extensions et sous-classes des modèles. Le chapitre comporte également des résultats sur la translation d'un modèle vers un autre et vice versa. Dans la dernière partie du chapitre, nous décrivons les structures utilisées pour le stockage et le traitement des observations collectées durant la procédure de surveillance, ainsi que le traitement des explications représentant leurs résultats.

Après cette introduction, dans le Chapitre 3, nous abordons directement le problème de surveillance de systèmes répartis temps réel. La solution décrite traite de l'approche basée sur un modèle comme mentionné dans l'introduction. Étant donnés le modèle d'un système sous surveillance et les observations en résultant, nous utiliserons la théorie des dépliages afin de définir lesdits dépliages avec contraintes. Intuitivement, un dépliage avec contraintes est une structure représentant les possibles scénarios composés d'événements produits par un système sous surveillance. Bien sûr, dans ce chapitre, nous introduisons l'aspect temporel et son influence sur les dépliages avec contraintes. La description des dépliages avec contraintes inclut les deux modèles suivants : les réseaux d'automates temporisés et les réseaux de Petri temporels. De cette façon, le lecteur peut suivre les différences concernant l'application des deux modèles et capturer des aspects communs relatifs à la problématique de la supervision. Dans ce chapitre, nous rappelons et étudions plus précisément quelques problématiques pouvant surgir durant la procédure de supervision basée sur notre approche. Ces dernières incluent : le problème des événements invisibles qui sera décrit plus loin et analysé dans la section suivante, ainsi que la non-monotonie de dépliages avec contraintes. Nous présentons également des études de cas pour les deux modèles.

Le Chapitre 4 est dédié au problème des systèmes avec boucle invisible et observation partielle. En fait, il est rare qu'il y ait une possibilité d'observation attentive concernant tous les événements se produisant dans le système. En dépit du fait que seule une partie des survenues d'événements puisse être physiquement vérifiée, il se peut simplement que trop d'entre eux soit surveillé, ou bien que les surveiller soit inutile. Mais il existe de nombreuses situations dans lesquelles les informations sur les événements invisibles dans le système soient hautement souhaitables, étant donné qu'elles affectent significativement la performance du système. Cependant, quand nous supposons l'existence de tels événements dans le système, qui est lui basé sur un modèle, nous devons nous préparer au fait que, lorsque nous recherchons des explications à certaines observations, nous puissions nous retrouver face à un problème avec des scénarios dans lesquels il puisse y avoir un nombre infini d'événements. Ceci est dû à la présence desdites boucles inobservables composées uniquement d'événements invisibles et non-surveillés. Le Chapitre 4 décrit étape par étape comment faire face à un tel problème, de

sorte que toutes les explications possibles puissent être stockées et tracées, même si leur nombre est infini. Le chapitre débute par la description du modèle le plus simple qui est en fait une machine à états finis, et termine avec un cas général de réseaux de Petri.

Dans le Chapitre 5, nous présentons de nombreux résultats associés à l'implémentation de solutions considérées dans notre travail. La première partie traite de certains problèmes relatifs à la surveillance basée sur le modèle de réseau d'automates temporisés. Nous abordons ensuite la partie liée au problème des boucles invisibles.

Le Chapitre 6 résume les résultats de notre travail et fournit des directions possibles pour les extensions futures.

Toutes les références utilisées dans ce rapport figurent dans la dernière partie du document.

Chapter 1

Introduction

1.1 Distributed real-time systems

Nowadays, hardly anyone has to be convinced that distributed systems are becoming an integral part of our everyday lives. Society is increasingly dependent on complex technology and it is not only about the quality of life but mainly about our security. All around us, we can find many examples of such systems, beginning from almost omnipresent mobile networks, and ending with flight control systems. Unfortunately, as shown by hitherto studies, what appears simple from the user perspective is no longer so obvious to the designer of such complex systems. Design and implementation of distributed real-time systems still are a big challenge for designers and developers.

The variety of applications that can be associated with the field of distributed systems is so large that the precise classification is a challenging task. Nevertheless, there are several basic divisions according to which distributed real-time systems are often classified. As we shall see, these classifications result from various functional and nonfunctional aspects of the concerned systems.

The first division which we mention is the division with respect to the deployment of elements of the system architecture. Basically there are two types of distributed systems: centralized and decentralized.

Centralized systems are largely based on client and server architecture. In this case, the server is a node that usually provides and performs services contracted by its clients. A centralized system has a hierarchical architecture, on top of which is a main management unit. As opposed to centralized systems, decentralized systems do not have a clear hierarchy with a parent node which is highlighted. Elements of a decentralized system are usually independent. They can of course communicate with each other, but as such, they generally make decisions independently of other system components. An example of decentralized systems which are very popular recently is peer to peer networks through which users can share their files. Usually in such

a network, each user only has partial information about the files which are available in the system. What is more, there is no single, global coordinator.

Another frequently cited criterion of division of distributed systems is the division into homogeneous and heterogeneous systems. Despite numerous advantages of homogeneous systems, when referring to very large and physically distributed systems, we will have to deal with heterogeneous systems. These include mobile networks, internet, large-scale computer networks (*e.g.*, popular in recent times GRID, [49]). The variety of software and hardware within such a system, which more or less is a complex network of different computers, entails many difficulties. One of them is that the operation of services and programs that cooperate with each other can significantly vary in different parts of it. In heterogeneous systems, there are also other problems that often do not apply to homogeneous systems. Scalability issue is one of them. The management of such systems is often difficult and requires the assistance of specialized software layer.

One of the key characteristics of distributed real-time systems is also time constraints which are assigned to some states or actions. For example, let us take a comprehensive system managing hospitals. These days, such systems are composed of many different modules which comprise almost all areas of an hospital activity: managing queues of patients, supporting the functioning of medical clinics, managing operational blocks, control of machines in laboratories, financial management of hospital, feeding patients, etc. Among such a large diversity of components and processes, there are many time constraints that apply to them. Some of them are strict and exceeding them can lead to disastrous consequences, as the death of the patient. Others are less restrictive and exceeding them does not cause serious problems. To distinguish these two types of time constraints, we respectively use the concept of hard and soft real-time systems. As an example of hard real-time systems we can give all the systems, in the case of which the passage of time plays a crucial role, such as systems that monitor patients' vital functions and allow for a relatively rapid response of other machines that improve the condition of the patient, or even call the hospital staff. A common example of a real-time system is a pacemaker.

As a very simple instance of soft real-time systems, we can provide a system for serving meals in the hospital. In this case, time does not play such a significant role, and in most cases, a meal served with a few minutes delay does not result in major consequences. A similar example may be the queue management system for patients. In this case the passage of certain scheduled time constraints in such systems also does not pose any significant consequences.

As mentioned, the modeling of distributed real-time systems often requires the use of various time constraints. Later in this document (see Chapter 2), we can note that many different kinds of time constraints were introduced along with many types of formal models. These restrictions dif-

fer among others in types of expressions which are used to formulate them. They range from simple linear expressions to, for example, more complex expressions with derivatives. It was shown many times that the complexity of these expressions directly affect the possibilities for analysis and verification of models that use them.

Note that time constraints are, in short, the functions that operate on the values of physical clocks which are available in the system. However, at this point, some difficulties may arise. Since the system is distributed, it can have multiple independent clocks. Imagine a situation in which the distributed system consists of several computers. Each computer has its own local clock which can be read or modified. When we now model the system along with its time constraints, we may find that a program on one computer wants to read the value of the clock on another computer. But, as it is known in the field of clock synchronization protocols, this is not an easy task. However, in practice the direct remote read operation of value of the clock is not so common. Usually this is due to the fact that there is an additional layer of software or hardware which is responsible for synchronizing all the clocks. This way, all computers have access to the virtual clock which can provide a common reference point. Of course, such a solution is not perfect since the clock synchronization is encumbered with some errors resulting *e.g.* from delays in communication between computers, from differences between the parameters of physical clocks (which have an influence on different frequencies of the clocks), and even from differences in the parameters of the environment in which computers are located (*e.g.* the ambient temperature).

Issues related to time synchronization protocols in computer networks have been fairly well studied. Particularly noteworthy is certainly one of the most popular synchronization protocols, namely, NTP[1] described *e.g.* in [74], which has become a widely used standard. Solutions used in the NTP show the complexity of the problem of synchronization of clocks in a distributed environment.

Unfortunately, not all uncertainties associated with the synchronization of clocks can be solved by means of clock synchronization protocols. Since the uncertainty time measurement in distributed systems is inevitable in many cases, it is natural that the problem also affects the formal models used to represent them. For this purpose, a number of formal models was introduced which directly address some aspects of the reliability of measurement of time in distributed environments.

An important characteristic of distributed systems is their unpredictability, non-determinism. For example, let us consider communication between components of a distributed system. It is usually asynchronous and it is often impossible to accurately estimate the delay after which data is received by its recipient, assuming that the data is received at all; breakdowns and

[1]Network Time Protocol

delays are nothing unusual in such large systems.

In the case of smaller systems such as multiprocessor systems with shared memory, communication is frequently synchronous. In this case, it is easier to plan some procedures and to predict the duration of their operations.

Modern architecture of distributed systems is largely based on a network of interconnected computers that can exchange data relatively freely. Each computer runs a number of programs. In general, each of these programs consists of one or more processes that can communicate with each other. In addition, frequently within the processes, we may find many threads which work concurrently.

As we can see, the communication and the concurrency can occur in distributed systems at several levels of architecture. On the one hand, it is a consequence of multiple users of such systems. On the other hand, the users can simultaneously initiate many tasks. An important factor is the geographical dispersion of the processing nodes. As it turns out, this factor is quite important especially in the context of the time constraints imposed on the distributed system. This is largely due to the communication time which strongly affects the overall speed of the information processing.

1.2 Supervision

When designing distributed real-time systems, one can not predict all the situations that may cause the malfunction of such systems. The fact that a system is distributed already hampers a lot of job to its designers and testers. Moreover, if we mention time constraints and the fact that the environment in which the system works can undergo constant changes, it appears that accurate static analysis of such a system is practically impossible. Often, it is impossible to analyze in such a system even so seemingly basic issue as reachability of states. At this point, the only possibility of verification of system is its monitoring during its operation, usually within the limited scope which is relatively possible to predict (*e.g.* in a given timeframe and with limited values of certain parameters). This is one of the reasons for which, in the further part of the book, we deal with selected aspects of supervision of distributed systems.

Before we proceed to the description of some details related to the supervision of distributed real-time systems, we introduce several terms which are used in our work. A supervised system is a system which is monitored in order to detect a certain set of behaviors which can cause for example a malfunction. A supervising (or monitoring) system is the whole of notions and elements which serves to monitor a supervised system, and then helps to analyze the collected observations in order to identify some events and behaviors.

In order to effectively monitor and diagnose a system, the monitoring

system must have adequate information about the events which take place. The decision about what kind of information is collected and analyzed by the supervising system usually belongs to the designer of the system. Of course, it is important for the information to be useful in detecting a potential target which is usually wrong and unintended operations of the system. Also note that gathering excessive amount of information in order to reproduce the actions of the system, in many cases, turn out to be pointless, especially when it concerns real-time systems. This follows from the fact that the operation of such systems is usually not fully deterministic, and frequently it is impossible to recreate the conditions in which the system worked.

How is information about the events contained in the observation produced? Basically there are two sources of origin of the events. The first one is based on mechanisms, procedures that are directly integrated in the supervised system. As an example, we can consider a program inside of which the programmer embedded special procedures which produce information about events. The second source of events is based on external mechanisms. Typically, these are hardware solutions that do not interfere with the operation of the monitored system. However, this solution is usually less flexible and more expensive.

How does a monitoring system collect and process information about events? A typical distributed system consists of many independently functioning devices which exchange their data from time to time. Observations produced by these devices in the process of supervision may be used in different ways. We can distinguish two basic scenarios: they may for example be stored locally in the form of local traces and wait for further processing; or they can also directly be sent to a central monitoring unit where they can be analyzed all together.

An advantage of the first case is that, sometimes, an unnecessary communication cost can be avoided if certain properties which are searched can be detected locally without referring to the whole system. However, it may happen that at some point, there is a need to send the results to some other monitoring unit in order to determine the global view of certain events that took place. Unfortunately, this can easily lead to a sudden transfer of large amounts of data.

In the second case, the observations are directly transmitted to the supervisor without a local processing. Also, this approach carries the risk of overburdening the system with a large number of observations sent to a supervisory system. However, in contrast to the first approach, the load of the system, in particular communication links, can be noticeable throughout the entire period when the system is active. Therefore, a reasonable solution seems to be a partial distribution of supervision, and thus avoids the disadvantages of centralized systems. However, note that this approach is not always possible, for instance because of the existing architecture of the system or a specificity of the monitoring.

In our work, we will focus on the specific case of monitoring which is called the model-based supervision. The general scheme of a model-based supervision system is shown in Figure 1.1. As we can see, in such a monitoring system, we can distinguish four principal elements:

- *model of the system* which is supervised. In other words, it is a formal representation of the monitored system which can be described *e.g.* by networks of timed automata or time Petri nets (for details see Chapter 2);
- *observations*, that is information about events that are produced by components of the system;
- *explanations* — a set of scenarios which is produced on the basis of the model of the system and observations. In general, an explanation is a collection of events with a relation of order between them. Since we consider timed models, there are also time constraints which are associated with explanations. The time constraints allow for assigning time-stamps to specific events in the explanations. It should be noted that there may be many different explanations for a single set of observations as many trajectories of a given model can produce the same set of observations;
- *monitoring module*: using the knowledge about the model of the system analyzes observations in order to obtain explanations of the system behavior.

Supervision of distributed real-time systems involves many difficulties. As mentioned above, monitoring is done during the operation of the supervised system. This makes the system interfere with the functioning of the monitoring system. As we can expect, this in turn causes a distortion of the results of monitoring, according to the known principle of uncertainty. This follows from the fact that the production of information about events which are then analyzed by the supervising system requires a certain amount of time. Unfortunately, while the operations that take place in the system are slowed down by the monitoring system, real time goes by. This in turn can greatly affect the results of the monitoring system. The second reason why the monitoring system may affect the functioning of the supervised system is the process of collecting information about the system events. Namely, a large amount of information about events occurring in the system and the use of the same transmission links for both the monitoring system and the monitored system mean that time required to transmit data used by the system can be considerably greater in comparison with the situation in which there is no supervisory system.

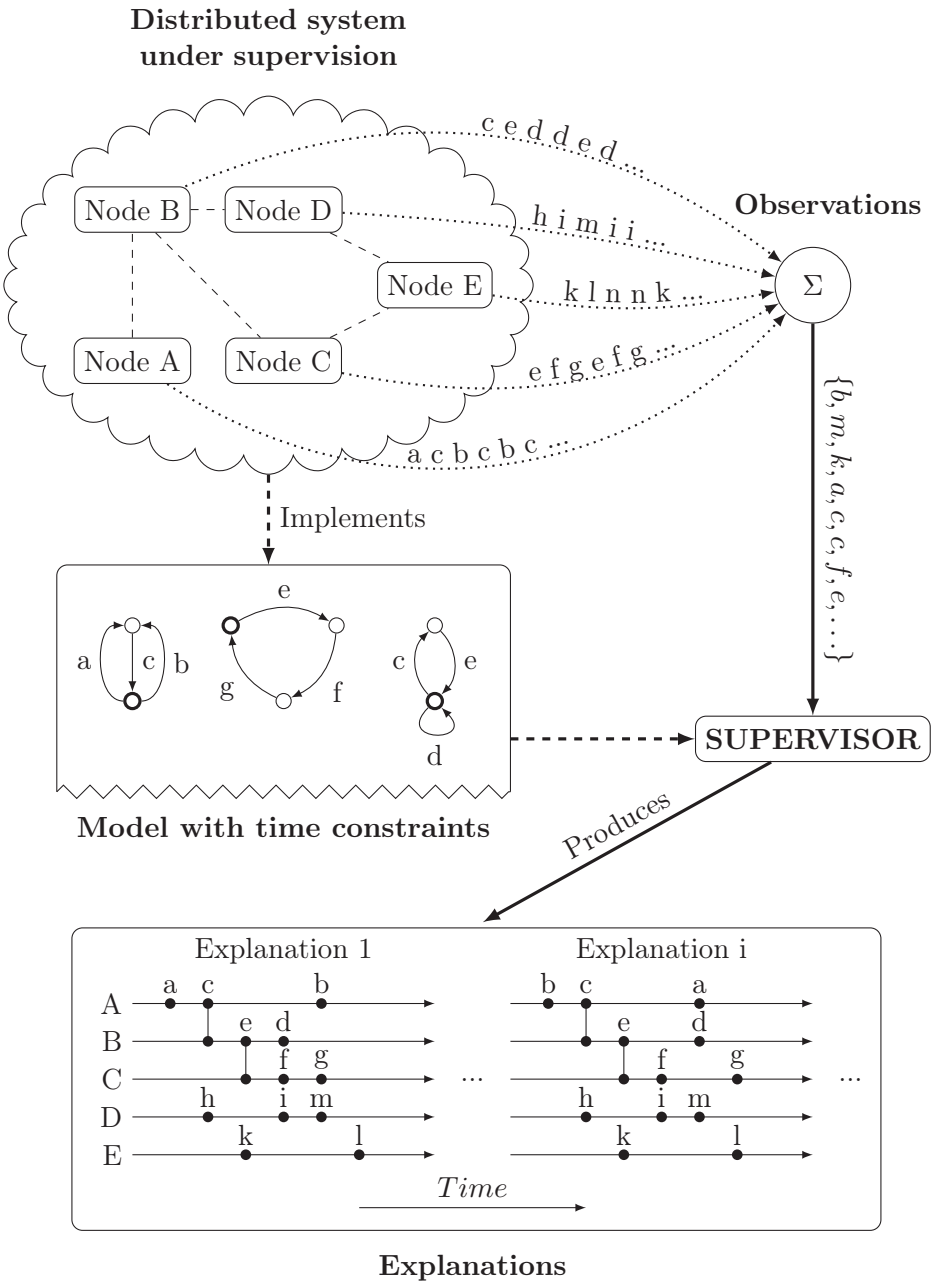


Figure 1.1: Model-based supervision of distributed systems

As mentioned earlier, a factor that hinders a monitoring real-time distributed system is the lack of global clock, in relation to which one could measure the time occurrence of each event.

As shown by numerous experiences of designers and programmers, the very fact that the system is based on restrictions relating to real time is a major challenge for monitoring systems. When we add to this the fact that we deal with a distributed system, the whole procedure of monitoring becomes considerably complicated. As we look at procedures for debugging programs or distributed programs, we find that they are relatively time-consuming, and they can significantly delay the work of the program that is debugged.

1.3 Choice of models

In the previous section, we described the various elements that are parts of the monitoring system which will deal with. One of the main assumptions of such a system is the existence of a model of the supervised system. Therefore, the question arises what kind of formalism should be used to model the system so that all requirements of the supervisory system are met. The literature on this subject gives us many different possibilities. Among them, we can find two recently very popular and frequently used models: networks of timed automata and time Petri nets. Which of these models is chosen depends on many factors such as, for example, the characteristics of the system which is supervised or properties that are monitored.

Both networks of timed automata and time Petri nets can model distributed systems with time constraints. In this case, an important aspect certainly is the possibility of modeling the parallel events.

For instance, imagine there is a distributed system in which there are many parallel processes. From time to time, these processes interact with each other. We assume that each of these processes consists of a sequence of actions. In this case, it seems quite natural to model each of the processes as an automaton. Additionally, in case of interaction between the automata, a simple synchronization of actions can be used.

To correctly model the distributed real-time system, the aspect of time must be taken into account. Both mentioned models provide the possibility of imposing time constraints on events and states that they generate. However, it should be noted that, for each of the models, these restrictions are different (more details in the next chapter.)

In the previous section, we briefly mentioned the role played by the process of acquisition of an observation in the supervisory system. We have identified the possible modalities of observation. Nevertheless, there is a question which remains, namely what form an information about events that occurred in the system should have so that it is useful for further analysis.

As we know, in distributed systems it is difficult to establish a common reference point for all events such as the time, for example. Even in situations where it happens that the supervisory system has information about the time of occurrence of certain events, usually they are not detailed enough to identify possible errors in the operation of the supervised system. Fortunately, even without the aid of physical clocks, the monitoring system can get another important information about events. Namely, using for example vector clocks ([48, 69]) we can explore the causal dependencies between events. In this way, two basic types of relationships between events can be distinguished:

- relation $a \leq b$ means that the event a causally precedes the event b;
- relation a co b means that the two events a and b are executed in parallel.

It may also happen that the causal relation between events is not known. For this purpose, we use the character ?. In other words, the expression $a?b$ means that causal relationship between a and b is an unknown. As we shall see, presented relationships play a fundamental role in describing the problems associated with distributed systems. The dependencies between events we introduced define the so-called partial order relation. Thus, the relation allows to describe the order of events in a distributed system and, as such, it is one of the central points in the structures used for storing and processing information about events.

Using the above relations in the case of observation, we can note that, at best, only on the basis of observations, we know the exact causal dependencies between all events. And, in the worst case the order of events is completely unknown. To learn the order of the events, we can use for this purpose a monitoring system which tries to compute the searched dependencies on the basis of the model of the system and observations .

In the remainder of the book, we will generally assume that the causal relation between events in the observations is unknown. Of course, in many situations, this implies a larger number of possible explanations consistent with a given observation. On the other hand, this approach shows the broad capabilities of the system of supervision, both from the theoretical side and the possibility of implementation.

In previous paragraphs, we shortly mentioned the concept of unfolding which is a structure that brings together the events occurring in the system and causal relationships between these events. It is characteristic of this structure that every event has a unique history in it. In other words, given any event from such a structure, we can clearly trace the process by which the event occurred. Since in the book we introduce the problem of monitoring in the context of the two models, we also introduce two similar ways to formalize explanations.

To represent the explanations produced by the system based on Petri nets, we use so-called occurrence nets. It is a structure that represents occurrences of events in the system and the states that accompany them. Since, as already mentioned, each event has its own unique past when the model contains time constraints, the situation becomes more complicated. Consider a simple situation in which we deal with an action that may occur within a specified period of time. If we would like to describe all possible occurrences of the action using a simple occurrence net, the number of corresponding events could be in the worst case infinite. For this purpose, we shall use the symbolic variant of occurrence nets whose structure groups events with execution time within a certain time interval. Thus, instead of enumerating all the events that can occur within a specified period of time, we can define a symbolic event with a symbolic time constraint which represents all possible realization times of the considered event.

We have a similar situation in the case of networks of timed automata. However, as we shall see in a later chapter, the structure used to represent events and states of networks of timed automata is slightly different. To explain shortly the difference, in occurrence nets each condition or event represents a single element of the associated model which, in our case, is a Petri net. On the other side, in the case of an event structure used for networks of automata, each element of such a structure may represent at once several elements of the associated model. It can group both transitions and locations of automata.

As we know, as a result of the monitoring process, we obtain a set of scenarios. In turn, these scenarios include events which reflect the possible behavior of the system, *i.e.* explanations. Obviously, explanations must be consistent with the observations that come out of the monitored system. For that purpose, we introduce the new concept of constrained unfoldings. As we will see, this will be nothing but an unfolding which includes events consistent with the observation. This structure is based on the notion of unfolding, with the difference that it takes into account the observation.

An introductory example

The example we present below is inspired by a real problem. Nevertheless, we present it in a very simplified version just to show the idea of the problems we deal with in the remainder of our work. We show how the concept of network of timed automata can be used in a simple production process. This type of problem can be seen as a simple variant of scheduling problem in which there are certain resources which must be appropriately managed. The system which we consider is depicted in Figure 1.2 and consists of three basic types of components: suppliers, a schedule and machines.

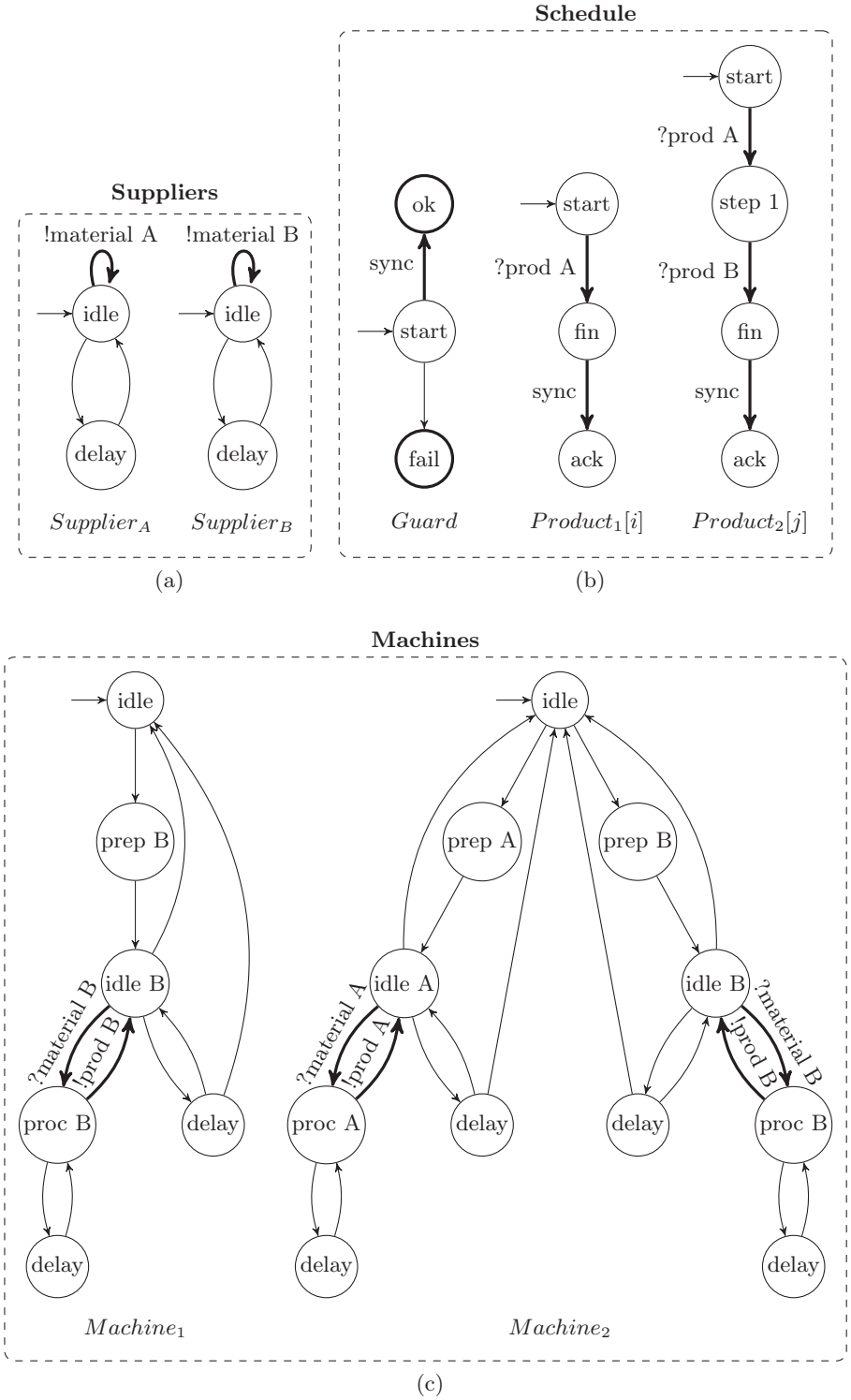


Figure 1.2: A simple production system

Each component contains several automata which all together form a network of automata. Each circle represents a state and each arrow represents a transition between states. All the arrows without any input state indicate the initial states. The arrows drawn in bold represent special transitions called synchronizations. Moreover, transitions concerning a synchronization have the same label. In our example, we have three types of synchronizations: *material X*, *prod X*, *sync*, where X is equal to A or B. For the sake of our example, we added two characters before some names of transitions, *i.e.* "?" or "!". In the case of synchronizations containing transitions with these characters, we assume that synchronization can involve only two transitions with the same name, one with "!" and one with "?" in the label. On the other hand, the synchronization *sync* denotes execution of all transitions with this name.

Below, we shortly describe all the components.

- *Suppliers* — their task is to provide raw materials which are then processed by machines into final products. We distinguish various types of materials. In our example, we use two types of them, *i.e.*: *material A*, *material B*. Depending on the sort of material, it can be processed by different types of machines. Sometimes, the suppliers may fail to deliver some materials causing some delays in production. Such situations may obviously have an impact on the production plan.

- *Schedule* — it shows how many details of each type should be produced in a given period of time. In other words, it represents deadlines which should be met. In the Figure 1.2b, we can observe three types of automata. Two of them represent a production process of two sorts of the final products. The first product $Product_1$ only needs one semi-product *prod A* to be finished, while $Product_2$ needs one semi-product *prod A*, and then one *prod B*. We assume that there is a certain number of both type of products to be made. When all the final products are finished (see the states with labels *fin* in Figure 1.2b), there is the last operation called *sync* which is used in order to confirm that the whole schedule is accomplished on time. The automaton that serves as the timer is called *Guard* and has two transitions: *ok* in case the deadline is satisfied, and *fail* otherwise.

- *Machines* — they are used to process the materials. Each machine can only process certain types of materials. In our example, $Machine_1$ only accepts materials of type B, whereas $Machine_2$ accepts both types of materials, i.e. A and B. Before a machine starts production, a certain amount of time is necessary to prepare it (states labeled with *prep X*). Then, it can one by one process all delivered materials and transform them into the semi-products. This procedure is shown in the form of a simple loop consisting of two transitions: *material X* and *prod X*. In our

example, we also consider some possible breakdowns of the machines which may cause significant delays (states labeled with *delay*). The important point is that we assume the delays are not always observable and can be difficult to detect during production process.

The system presented in the example is static in the sense that there is a given number of components which does not change during its activity. We mentioned in the description of the components some time constraints related to some transitions and states. Having the information about time constraints in the model is very important as they make model more realistic and finally reflect the possible problems with more precision.

The system we present is quite simple but, even on the example of such a system, we can already analyze a few issues which may arise in the context of monitoring systems with time constraints. For example, we mentioned earlier that there could be various types of failures in the system that cause delays. The system user may for instance ask the following questions: in the case of machine failure, how long it takes to repair it so that the plan is completed on time and that the production does not have to be shifted onto another machines; or if time given for reparation is exceeded, is shifting of the production onto other machine sufficient to satisfy the constraints in the schedule; if delays occurred, at which point of the production process they happened.

1.4 Positioning and contribution

In our work, we decided to base our solution on the theory of unfoldings which we found the most suitable for the objective we wanted to achieve. This theory was initially introduced by McMillan in [70] and then extended and improved, notably by Esparza in [42]. Originally, the unfoldings were introduced to deal with the model checking as an alternative, especially for the methods based on an interleaving semantics. Unfoldings have shown to be more concise when it comes to memory requirements. They also enable us to store information in the form of a partially ordered event structure, whereas the structure based on fully sequential structures can not store information about parallel events. One of the main compendiums which tries to synthesize the principal knowledge about unfoldings was published by Esparza and Heljanko ([44]).

The theory of unfoldings found some applications soon after its introduction. The one which was especially in the field of our interests was diagnosability of distributed systems described by *e.g.* [15]. Supervision of distributed real-time systems that we present later in the book is a continuation of this work. The subject of diagnosis of distributed systems was described in [46]. However, this work does not consider the time constraints which often play a key role in monitoring distributed real-time systems.

For example, the problem of unfoldings and time constraints in the context of Petri nets has been studied in [34, 32]. In these works, a new unfolding technique for time Petri nets and its exemplary use for diagnostic purposes were proposed. This structure differs from the typical unfoldings of Petri nets, inter alia, that it contains information about time constraints. Unfortunately, when we take into consideration an unfolding of time Petri net and remove all the time constraints from it, we may find that the remaining structure is substantially greater than the unfolding built on the basis of the same Petri net but without time constraints. This is caused by duplication of some events in order to assign to them the relevant time constraints. The problem of duplicates was solved, for example in [85, 84]. This new **construction method of unfoldings of time Petri nets**, proposed in these works, solves the problem of multiple events at the expense of more complex time constraints. As we will later note in the book, unfoldings constructed in this way have an important advantage over the first approach. Namely, if we remove the time constraints from the structure, it turns out that its size is not greater than the size of the unfolding of the untimed version of the model. This enables us to split, if necessary, the construction of unfoldings into a number of stages. First, we construct an unfolding of an underlying untimed version of the model. Then we add the appropriate time constraints. Note that no new events are added to the structure after the first stage.

One of the major part of our work is the study about **unfoldings of networks of timed automata used to perform supervision**. For the first time, the concept of such unfoldings was proposed in [29, 26]. We decided to extend this work and to adapt it to the problem of monitoring of distributed systems. In [52], this led us to a technique of construction of unfoldings guided by an observation and a method to infer the possible dates of occurrences of actions by using symbolic constraints in networks of timed automata. This was also the beginning of our further work on **constrained unfoldings** which is the name for this type of unfoldings. It resulted in a set of algorithms dedicated to the unfoldings of networks of timed automata, and their implementations.

In Chapter 4, we deal with the problem of unobservable events, *i.e.* events that are reflected and visible in a given model, but that can not be observed during the operation of the associated system. The problem of partial observation already appeared many times in the literature and was discussed in the context of many different formal models, for instance in the domain of Petri nets ([37]) and automata ([19, 27, 41, 18]). The issue was also discussed, *e.g.* in the works on model checking ([43]), or supervisory control ([28]). However, a large portion of those events that are invisible are ignored and there is no direct information about them. One example of such an approach is supervision with observers described *e.g.* in [28]. Intuitively, in order to create the observer, a model of system is taken and then all the transitions corresponding to unobservable events are removed. As a result,

the states between which were unobservable transitions are grouped all together. This creates new states that group original states. It may be noted that, in the case of the observer so defined, there is no direct information about events that may occur in the system, but that can not be observed. Moreover, in this case, an additional drawback is the loss of some information about relations of causality between the events. For instance, it is not possible to determine whether some events were executed in parallel.

Another approach to the problem of unobservable events was also proposed by Wimmel in [88]. In his work, he proposed to remove the invisible transitions in such a way that the modified model was pomset-equivalent. Intuitively speaking, causal relations are maintained between the events. However, the case studied by Wimmel is limited to a specific case of Petri nets. As far as we know for the general case of Petri nets, it appears that the problem still does not have a solution.

To conclude, information about unobservable events can be crucial to detect any abnormal operation of the system, such as hidden delays that can often creep into the system and which are impossible to track by the monitoring system. Thus, removing such events may not be desirable. However, a problem remains as some unobservable events in the system can be repeated and theoretically produce infinite unobservable loops. For this reason, in Chapter 4, we propose a new approach which consists in **unfolding a Petri net guided by a partial observation**. This type of so-called **partial unfolding** enables us to store and process information about infinite and unobservable loops using a finite amount of memory. In the chapter, we deal with several classes of Petri nets showing that the complexity of the model influences the complexity of the problem. This part of the book has not been published yet.

In the last part of our work, Chapter 5, we focus on the **implementation of proposed solutions**. So far, relatively few universal tools were created, based on the theory of unfoldings which could be easy to compare in terms of usability and performance. Among existing tools, we can for example find the PEP tool (Programming Environment based on Petri Net; see [83, 54]), recently Romeo (a tool for time Petri nets analysis; see [50, 2]) which we also mention later in our work, the Model-Checking Kit which is a collection of programs including some verification algorithms based on the theory of unfoldings ([81]). Among various applications of unfoldings, we can find the aforementioned work on the diagnosis of discrete event systems ([15]), works on unfoldings used in the context of asynchronous logic circuits ([61, 62]), as well as the verification of mobile systems ([73]).

During our research, we implemented and verified algorithms for **unfolding networks of timed automata in the presence of observations**. In cooperation with the Institut de Recherche en Communications et Cybernétique de Nantes which is the author of the Roméo tool, we performed tests on a **construction of constrained unfoldings for parametric time Petri**

nets. We also created an experimental tool which we used to compute **constrained unfoldings of Petri nets with partial observation** described in Chapter 4.

Moreover, we decided to reengineer and improve the tool for unfolding networks of timed automata. This way, **the Spinta tool** was created. It can operate on networks of automata with parameters and systems of linear constraints, *i.e.* polyhedra.

All the tools mentioned above were used, for example to prepare the case studies and examples presented in this book.

1.5 Organization of the document

Chapter 2, begins with a brief presentation of the basic notions which we use in the following chapters. Section 2.1 is a short introduction to timed transition systems which are a reference point for a further discussion about other models. Then, sections 2.2 and 2.3 present two different families of popular models aimed at modeling distributed real-time systems, namely networks of timed automata and time Petri nets. After the introduction of the models, we shortly describe in Section 2.4 basic results about the decidability of various problems in the context of both models. We very briefly present some extensions and subclasses of the models. The chapter also contains some results on translation of one model to another and vice versa. In the last part of this chapter, we describe the structures used for storage and processing of the observations collected during the monitoring process and the explanations which are its result.

After this introduction, in chapter 3, we directly go to the problem of monitoring of real-time distributed systems. The described solution is based on the model-based approach as mentioned in the introduction. Given a model of a monitored system and observations that come out of it, we will use the theory of unfoldings in order to define so-called constrained unfoldings. Intuitively, a constrained unfolding is a structure which represents possible scenarios consisting of events that are produced by a supervised system. Of course, these scenarios must be as consistent as possible with the observations obtained from the supervised system. Additionally, in this chapter, we introduce the aspect of time and its influence on constrained unfoldings. The description of the constrained unfoldings includes the two following models: networks of timed automata and time Petri nets. This way, the reader can follow differences in application of the two models and capture some common aspects related to the issue of supervision. In the chapter, we recall and closer look at a few issues that may arise during the process of supervision based on our approach. These includes: the problem of invisible events which will be further described and analyzed in the next section, non-monotonicity of constrained unfoldings. We also present some

case studies for both models.

Chapter 4 is dedicated to the problem of invisible loop systems with partial observation. In fact, it is rare that there is a possibility of careful observation of all events occurring in the system. Despite the fact that not all occurrences of events can be physically verified, it may also simply turn out that either there are too many of them to monitor or monitoring them is unnecessary. But there are many situations where information on invisible events in the system is highly desirable, since it significantly affects the system performance. However, when we assume the existence of such events in the system, which is based on a model, we must be prepared for the fact that, when searching for explanations of some observations, we encounter a problem with scenarios in which there may be an infinite number of events. This is due to the presence of so-called unobservable loops only consisting of the invisible, non-monitored events. Chapter 4 step by step describes how to cope with such a problem, so that all possible explanations could be stored and tracked, even if their number is infinite. The chapter begins with the description of the simplest model which is a finite state machine, and finishes with a general case of Petri nets.

In Chapter 5, we present several results associated with implementation of solutions considered in our work. The first part presents some issues related to the monitoring based on the model of network of timed automata. Then we move on to the part related to the problem of invisible loops.

Chapter 6 summarizes the results of the work and provides possible directions for future extensions.

All the references used in the book are in the last part of the document.

Chapter 2

Models

Before we move on in subsequent chapters to discuss the main problem which is supervision of the distributed real-time systems, we introduce the necessary set of notions that will serve us in the rest of the work.

As a basis of our further discussions, we describe formalisms which we use to represent various types of distributed systems. Among numerous formal models that have been defined in recent years, we have chosen two of them, which are still very popular and are used in various applications. Namely, these are mentioned in the introduction: networks of timed automata and Petri nets.

Both models proved to be grateful material, not only for researchers but also for practitioners. And even though they do not necessarily have all the desirable characteristics in their basic versions, they represent a good starting point for a variety of more specialized and complex models. Moreover, the potential range of applications of such models is enormous. Along with new research results, many new applications or solutions are created and often replace their older, worse counterparts.

Although these models were introduced a relatively long time ago, they have been developed quite independently of each other in the context of distributed systems. However, recently we can observe a trend towards unification of the two models. In this way, many problems solved by using one of the models could be transferred to the ground of the second one.

The chapter starts by introducing the concept of transition systems which, due to its characteristics, will provide a reference point for models based on networks of timed automata and time Petri nets. Then, we discuss properties of network of timed automata, starting with its most basic version which is a finite state machine, and ending with a model consisting of a number of finite state machines and time constraints. Analogically, we discuss the model of Petri nets and its variants with time constraints and parameters which gives a model known as a parametric time Petri net.

Then, we briefly describe the basic properties of both models. These

properties are bounded to a number of algorithmic problems, which implies decidability questions about them. Above all, we can find among them the problems associated with verification of some basic properties of models such as state reachability, liveness, the possibility of deadlocks, etc. We mention briefly issues related to conversion between the models. This is largely due to an issue which was mentioned in the previous paragraph, namely whether problems described by one model can be automatically expressed using another one.

The chapter closes with a description of the formalisms used to describe two key concepts which are used in the context of monitoring distributed systems, *i.e.*: observations and explanations. In this part, we describe structures used to represent and process information which is critical to the supervisory system.

General notations

We denote by $\mathbb{N}$ the set of non-negative integers, by $\mathbb{Q}$ the set of rational numbers and $\mathbb{R}$ the set of real numbers. For $A \in \{\mathbb{Q}, \mathbb{R}\}$, $A_{\geq 0}$ (respectively $A_{>0}$) denotes the subset of non-negative (respectively strictly positive) elements of A.

Given $a, b \in \mathbb{N}$ such that $a \leq b$, we denote by $[a..b]$ the set of integers greater or equal to a and less or equal to b. For any set X, we denote by $|X|$ its cardinality. In the symbolic expressions, $\wedge$ denotes the logical conjunction, $\vee$ the logical disjunction and $\neg$ the logical negation operators. We will also use $\Rightarrow$ as the logical implication.

For a function f on a domain D and a subset C of D, we denote by $f_{|C}$ the restriction of f to C.

Let X be a finite set. A (rational) *linear expression* on X is an expression of the form $a_1x_1 + \cdots + a_nx_n$, with $n \in \mathbb{N}$, $\forall i, a_i \in \mathbb{Q}$ and $x_i \in X$. The set of linear expressions on X is denoted $Expr(X)$. A *linear constraint* on X is an expression of the form $L_X \sim b$, where L_X is a linear expression on X, $b \in \mathbb{Q}$ and $\sim \in \{<, \leq, \geq, >\}$. We will also use abbreviations like $=$ and $\neq$.

For the sake of readability, when non-ambiguous, we will "flatten" nested tuples, *e.g.* $\langle\langle\langle B, E, F\rangle, l\rangle, v, \theta\rangle$ will be written $\langle B, E, F, l, v, \theta\rangle$.

2.1 Timed transition systems

A timed transitions system enables us to describe a system that, during its operation, performs two types of operation: continuous and discrete. As we will see later, using timed transition systems, we can express semantics of both networks of timed automata and time Petri nets. This, in turn, enables us in some way to make a comparison of both models. The exact definition of timed transition systems is as follows:

Definition 1. ***(Timed transition system)*** A *timed transition system* (TTS) defined over a set of actions Σ is a tuple $\mathcal{T} = \langle S, S_0, \rightarrow, \Sigma, F, R \rangle$ where: DEF

- S is a set of *states*,
- $S_0 \subseteq S$ is the set of *initial state*s. If S_0 contains only one element we directly use the name of the element instead of the set;
- $\rightarrow \subseteq S \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S$ is the *transition relation*. For each transition $s \rightarrow s'$, s represents its *source* state, and s' is its *target* state,
- Σ is a finite set of actions disjoint from $\mathbb{R}_{\geq 0}$,
- $F \subseteq S$ is the set of *final states*,
- $R \subseteq S$ is the set of *repeated states*.

Moreover, the transition relation consists of two types of actions:

- delay transitions $s \xrightarrow{d} s'$, with $s, s' \in S$ and $d \in \mathbb{R}_{\geq 0}$, or
- discrete transitions $s \xrightarrow{a} s'$, with $s, s' \in S$ and $a \in \Sigma$.

In most cases, we skip the sets F and R, which means that $F = R = S$.

♦

In the first part of the definition, we can see the standard features of a transition system such as a set of states, a set of initial states, and a relation that defines transitions between the states. Then, the definition distinguishes two types of transitions that we mentioned earlier: continuous transitions, also called delays (as they refer to time), and discrete transitions, which represent the immediate change in the system. Note that in both cases, the transitions are marked with labels. For a discrete transition $s \xrightarrow{a} s'$ the label $a \in \Sigma$ denotes the name of the action. However, in the case of a continuous transition $s \xrightarrow{d} s'$, $d \in \mathbb{R}_{\geq 0}$ is the amount of time after which the system changes its state from s to s'.

It is worth mentioning that the transition relation often requires some standard properties (for more details see *e.g.* [75]) such as:

- *Time determinism*: if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$, then $s' = s''$;
- *Time additivity*: if $s \xrightarrow{d} s'$ and $s' \xrightarrow{d'} s''$, implies $s' \xrightarrow{d+d'} s''$;
- *0-delay*: $s \xrightarrow{0} s$;
- *Continuity*: if $s \xrightarrow{d} s'$ then for each d' and d'', such that $d = d' + d''$, there exists s'', such that $s \xrightarrow{d'} s'' \xrightarrow{d''} s'$.

An operation of timed transition system can be defined as a finite or infinite sequence of movements in which discrete transitions are interwoven with continuous transitions. Such a sequence of length $n \geq 0$ is called a *run* and takes the following form:

$$\rho = s_0 \xrightarrow{d_0} s'_0 \xrightarrow{a_0} s_1 \xrightarrow{d_1} s'_1 \xrightarrow{a_1} s_2 \cdots s_n \xrightarrow{a_n} s'_n \cdots$$

Sometimes, for a finite run, we can write $s \xrightarrow{d_0 a_0 d_1 a_1 \cdots d_n \cdots} s'$. If there is a run ρ such that the first state is s and the last one is s', we write $s \xrightarrow{*} s'$. A run is *initial* if its first state belongs to S_0.

A timed transition system can generate *traces*. For a run ρ, the trace of ρ is defined by $\mathsf{trace}\,(\rho) = (a_{i_0}, d_0 + \ldots + d_{i_0}) \cdots (a_{i_k}, d_0 + \ldots + d_{i_k}) \cdots$ where $a_{i_k} \in \Sigma$ for $k \in \mathbb{N}$. So defined trace forms a *timed word*.

DEF **Definition 2.** ***(Timed word)*** A *timed word* w over a finite alphabet Σ is a finite or infinite sequence

$$w = (a_0, d_0)\,(a_1, d_1) \ldots (a_n, d_n) \ldots$$

such that for each $i \geq 0, a_i \in \Sigma, d_i \in \mathbb{R}_{\geq 0}$ and $d_{i+1} \geq d_i$. Note that d_i is an absolute date. ♦

Given a timed word $w = (a_0, d_0)\,(a_1, d_1) \ldots (a_n, d_n) \ldots$, its untimed part can be extracted, *i.e.* $\mathsf{untimed}\,(w) = a_0, a_1 \ldots a_n \ldots$, and its duration $\mathsf{duration}\,(w) = \sup_{k \geq 0} d_k$.

A run ρ is *accepting* if:

- either ρ is finite and initial and its last state belongs to F, or
- ρ is an infinite initial run and there is a state $s \in R$ that infinitely often appears in ρ.

A timed word w is accepted by $\mathcal{T}$ if there is an accepting run ρ such that $\mathsf{trace}\,(\rho) = w$. Finally, the *timed language*, $\mathcal{L}\,(\mathcal{T})$, accepted by $\mathcal{T}$, is the set of timed words accepted by $\mathcal{T}$.

2.2 Networks of timed automata

In order to understand the notion of network of timed automata, we present it in two parts. In the first part, we introduce basic notions of a finite automaton and network of automata. Then, we add time constraints to the model. This way, as a final result, we obtain a network of timed automata.

2.2.1 Finite state automaton

Below, we show a classical definition of a finite state automaton which is a basic element of networks of automata presented in the remainder of our work.

Definition 3. ***(Finite state automaton)*** A *finite state automaton* (or *finite state machine*) $\mathcal{A}$ is defined by a tuple $\mathcal{A} = \langle L, l_0, T, \Sigma \rangle$ where: DEF

- L is a finite set of locations[1],
- $l_0 \in L$ is the initial location,
- $T \subseteq L \times \Sigma \times L$ is the set of transitions. Each transition represents a change between two states. A transition is denoted by $t = (\alpha(t), \lambda(t), \beta(t))$, which means that the transition t is labeled with $\lambda(t)$, its starting location is $\alpha(t)$ and its final location is $\beta(t)$,
- Σ is an alphabet of actions (a finite set of symbols). ♦

In the definition above, we can see that, like we did in the case of timed transition system, transitions can be attributed to some symbols of the alphabet Σ. For that reason, this type of automaton is also commonly referred to as *labeled* automaton.

It is noteworthy that there exists some other similar definitions of finite state automata. For example, there is a popular notion of acceptor finite state machine which distinguishes an additional set of final states. Every time one of the final states is reached, it means that a procedure consisting of a sequence of transitions was finished successfully. However, the definition presented above is sufficient to consider the problems raised in this book. Therefore, we confine ourselves to this definition.

Basically, automata can be divided into two subgroups: deterministic and non-deterministic automata. We say that an automaton is deterministic if, for any input state, there is at most one possible transition which can be executed. Otherwise, if for a single input state there is more than one possible transition, the automata is non-deterministic. For a finite state automaton, every non-deterministic automaton can be transformed into a deterministic automaton which accepts the same language, *i.e.* a set of words accepted by automaton. In the rest of the book, we assume that the models we use are non-deterministic. In this context, the solutions that we present are quite universal.

Example 1. An example of a graphical representation of a finite state EXM

[1]When we consider a finite state machine, notions of locations and states can be used interchangeably. Be aware that in the case of timed automaton and its extensions, a state frequently stores more than a location. It can for example contain information about time.

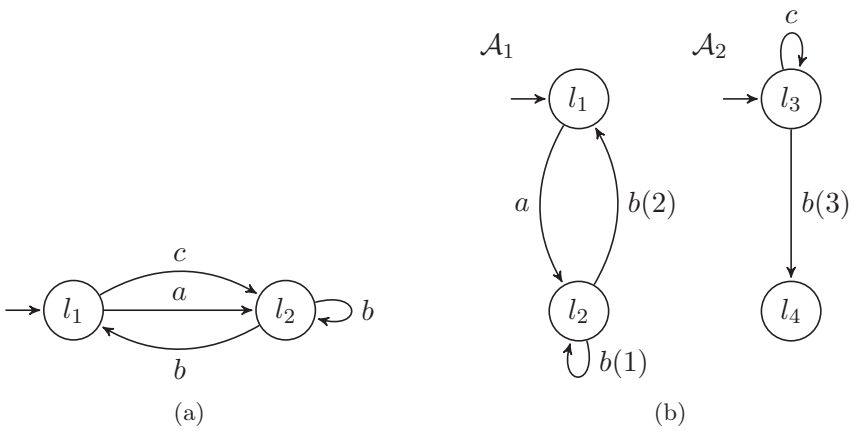


Figure 2.1: A finite state automaton (2.1a) and a network of finite automata (2.1b).

automaton is given in Figure 2.1a. l_1 is the initial state. There are three types of actions: a, b, and c. Note that the same action can be associated with more than one transition in the model. Thus, in this figure we can observe two transitions labeled by the same action b.

Note that each execution of such an automaton can be represented as a sequence of actions with the initial state in the beginning. However, when we consider a simple word of two actions cb, we can observe that the word can represent two possible executions. That is because of action b which introduces non-determinism into the model.

2.2.2 Network of finite state automata

Once we have a notion of a single finite automaton, we can take more than one of them and construct a more complex system. Such a complex system provides us with a possibility to model a distributed system in which events can appear in parallel. Network of automata can naturally provide independent functional elements of a distributed system where a single automaton reflects a single element of the system. Below, we present a formal definition of network of finite state automata.

DEF **Definition 4.** ***(Network of finite state automata)*** A network of finite state automata is a set $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ of n finite state automata with $\mathcal{A}_i = \langle L_i, l_{0i}, T_i, \Sigma_i \rangle$. We note $\Sigma = \bigcup_i \Sigma_i$ the set of all action names. ♦

The activity of the automata is synchronized on transitions having the same label. Formally, we define the set of synchronizations $Sync$ as the set of $(t_1, \ldots, t_n) \in (T_1 \cup \{\bullet\}) \times \ldots \times (T_n \cup \{\bullet\})$ such that $(t_1, \ldots, t_n) \neq (\bullet, \ldots, \bullet)$ and there exists $a \in \Sigma$ such that $\lambda(t_i) = a$ for every $t_i \neq \bullet$, and $a \notin \Sigma_i$ for every $t_i = \bullet$.

A global state of the network consists of current locations of all the automata. We note it by the vector $\vec{l}$, thus $\vec{l} \in L_1 \times \ldots \times L_n$. The initial global state is $(l_{01}, \ldots, l_{0n})$.

Definition 5. ***(Semantics of NA)*** Let $\mathcal{N} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ be a network of automata, where $\mathcal{A}_i = \langle L_i, l_{0i}, T_i, \Sigma_i \rangle$. The semantics of $\mathcal{N}$ is a labelled transition system $\langle Q, q_0, \rightarrow, \Sigma \rangle$ where: DEF

- $Q = L_1 \times \ldots \times L_n$,
- $q_0 = (l_{01}, \ldots, l_{0n})$,
- the transition relation $\rightarrow$ is defined by:
  - action transition: $\vec{l} \xrightarrow{a} \vec{l'}$ iff there exists $(t_1, \ldots, t_n) \in Sync$ such that $\forall i \leq n$, $\begin{cases} l'_i = l_i \wedge t_i = \bullet & \text{if } a \notin \Sigma_i \\ t_i = (l_i, a, l'_i) & \text{otherwise} \end{cases}$ ♦

Example 2. An example of such a network with two automata is given in Figure 2.1b. The set of actions consists of three symbols $\{a, b, c\}$. To distinguish the transitions with the same symbol, we put an extra number between parenthesis next to each of the symbols. The set of synchronizations is as follows $Sync = \{(a, \bullet), (\bullet, c), (b\,(1), b\,(3)), (b\,(2), b\,(3))\}$. Thus, a and c are local actions, whereas b requires synchronization of the two automata. EXM

2.2.3 Timed automata

Timed automata were introduced in [6] as an extension of classical automata. The first timed automata were equipped with perfectly synchronized clocks.

This gives a possibility to model time which is very important for many practical reasons. As we can expect, there are many algorithms for which right timing is crucial and determines whether it works correctly or not. The timed automaton introduced in [6] has been modified many times since its creation. Many of its different variants were analyzed, for example in the context of number of clocks, or a type of time constraints. Below, we will try to bring closer all basic aspects of timed automata and show a bit of its different variants.

Before we define timed automata, we introduce several notions which are required to add time into the model.

First, let us define a set X of clocks, each of which is a variable taking a value in $\mathbb{R}_{\geq 0}$. A valuation of the clocks is defined by a function $v : X \rightarrow \mathbb{R}_{\geq 0}$. Thus, if we want to read a value of a clock x, we use $v\,(x)$.

$\mathcal{C}\,(X)$ is a set of conjunctions of constraints of the form $x \bowtie c$, where $x \in X$, $c \in \mathbb{R}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. $\mathcal{C}_<\,(X)$ is a set of conjunctions of constraints of the form $x < c$ or $x \leq c$. A valuation v satisfies the atomic

constraint $x \bowtie c$ iff $v(x) \bowtie c$. This interpretation naturally extends to more general constraints.

The formal definition of syntax of a timed automaton is as follows.

DEF **Definition 6.** ***(Timed automaton)*** A *timed automaton* is a tuple $\mathcal{A} = \langle L, l_0, X, \Sigma, T, Inv \rangle$ where:

- L is a finite set of locations, also called control states,
- $l_0 \in L$ is the initial location,
- Σ is a finite alphabet of actions,
- X is a finite set of clocks,
- $T \subseteq L \times \mathcal{C}(X) \times \Sigma \times 2^X \times L$ is a finite set of transitions. A tuple $t = (\alpha(t), \gamma(t), \lambda(t), \rho(t), \beta(t))$ represents a transition from the state $\alpha(t)$ to the state $\beta(t)$, labeled by the action $\lambda(t)$, with guard $\gamma(t)$ and a set of clocks $\rho(t)$ reset by t,
- $Inv : L \to \mathcal{C}_<(X)$ assigns an invariant to each location. ♦

As we can expect, to describe the current state of timed automaton, we need to know at least two elements: the active location and the value v of all clocks in X. In other words, the state of timed automaton is a pair $(l, v) \in L \times \mathbb{R}^X_{\geq 0}$. Semantics of timed automata can be described using a previously described timed transition system and is as follows.

DEF **Definition 7.** ***(Semantics of timed automaton)*** Let us take a timed automata $\mathcal{A} = \langle L, l_0, X, \Sigma, T, Inv \rangle$. The semantics of $\mathcal{A}$ can be defined as the timed transition system $\langle Q, q_0, \to, \Sigma \rangle$ where:

- $Q = L \times \mathbb{R}^X_{\geq 0}$,
- $q_0 = (l_0, v_0)$ with $v_0(x) = 0$ for every $x \in X$,
- the transition relation $\to$ is composed of:
  - action transition: $(l, v) \xrightarrow{a} (l', v')$ iff
    $$\exists (l, g, a, r, l') \in T, \begin{cases} v \models g & (1) \\ v' = v[r] & (2) \\ v' \models Inv(l') & (3) \end{cases}$$
  - delay transition: if $d \in \mathbb{R}_{\geq 0}$, $(l, v) \xrightarrow{d} (l, v + d)$ iff $v + d \models Inv(l)$. Let us note that given the form of invariants, $v + d' \models Inv(l)$ for every $0 \leq d' \leq d$.

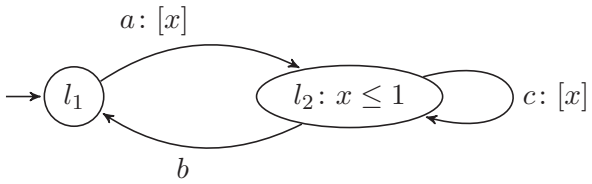


Figure 2.2: Timed automaton

The system operation begins with the state in which the initial location is active, and the values of all clocks are equal to 0. During operation of timed automaton, there are two types of actions that take place alternately. By analogy with timed transition systems from Definition 1, there are continuous transitions – related to the passage of time, and discrete transitions – associated with the change of location in the machine. In the case of discrete transitions, there are three conditions highlighted in the definition, which must be met: 1) the guards of the transition g have to be satisfied, 2) the clocks in r have to be reset to 0, and c) the invariant $Inv\left(l'\right)$ assigned to the new location has to be satisfied. In the original model defined in [6], it is assumed that the values of all the clocks change synchronously with the same frequency. It should be noted that none of the clocks can be stopped at any time, *i.e.* the time progresses continuously without any interruptions.

An important feature of models of this type is the property of urgency. This means that the model designer may impose time constraints, *e.g.* forcing the system to change the state at a particular moment of time or blocking the possibility of transition to a new state during a specified period of time.

Example 3. Let us consider the timed automaton in Figure 2.2. It has two clocks x, y. Consider an exemplary fragment of scenario representing an execution of timed automaton: $(l_1,(0,0)) \xrightarrow{2} (l_1,(2,2)) \xrightarrow{a} (l_2,(0,2)) \xrightarrow{1} (l_2,(1,3)) \xrightarrow{c} (l_2,(0,3)) \xrightarrow{0.5} (l_2,(1,4)) \xrightarrow{b} (l_1,(1,4)) \ldots$. As previously defined, the state of timed automaton is described by a pair consisting of its current location and the vector containing the values of all clocks. In this case, we have two clocks x, y. Thus, pair $(2,3)$ means that the value of the clock $v\left(x\right) = 2$, and the value of the clock $v\left(y\right) = 3$. EXM

Note that, on this basis, we can specify a timed word which we defined when describing timed transition systems. Having the sequence, of events we can convert it to the following sequence: $\left(l_0,(0,0),t_0\right) \xrightarrow{a} \left(l_0,(0,2),t_1\right) \xrightarrow{c} \left(l_0,(0,3),t_2\right) \xrightarrow{b} \left(l_0,(1,4),t_3\right) \ldots$, where $t_i \in \mathbb{R}_{\geq 0}, t_0 = 0$ and $t_i \leq t_{i+1}$ for every i. As we can observe, states produced as a result of delays were removed from the original sequences. While the other states which arose directly as a result of discrete event remained. In addition, each of these states was

extended with additional information about absolute time of occurrence of an event that produced the state, that is the time counted from the beginning of the operation of the automaton. In our case: $t_1 = 2, t_2 = 3, t_4 = 4$. In other words, by taking the difference $t_{i+1} - t_i$, we obtain the delay between successive discrete movements. Considering all these data, the timed word which corresponds to the sequence of events from our example takes the form: $(a, 2)\,(c, 3)\,(b, 4)$.

2.2.4 Network of timed automata

Given the notions of network of automata and timed automata, we can move to a model that combines the key features of both these models, namely the network of timed automata.

DEF

Definition 8. ***(Network of timed automata)*** A *network of timed automata (NTA)* is a set $\mathcal{N} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ of n timed automata with $\mathcal{A}_i = \langle L_i, l_{0i}, X_i, \Sigma_i, T_i, Inv_i \rangle$.

We make the assumption that clocks are *not shared* between automata, i.e. $\forall \mathcal{A}_i, \mathcal{A}_j \in \mathcal{N}$, $X_i \cap X_j \neq \emptyset$. The set of synchronizations *Sync* is defined as in the untimed case in Definition 4. We note $X = \bigcup_i X_i$ the set of all clocks. ♦

The state of network of timed automata consists of a mix of information about all the active locations and values of all the clocks. Thus, it is quite similar to the definition of timed automaton except the fact that there are now several automata instead of one. In the definition we additionally ensure that all the valuations of the clocks satisfy the relevant invariants assigned to the current locations. The initial state consists of all the initial states of all the components.

As we can expect, there are two types of transitions as it is defined for timed transition systems, *i.e.* action transitions and delay transitions. We assume that an action transition can also represent a synchronization. For this reason, we do not distinguish these two notions below. In the first case, in order to execute an action transition $\left(\vec{l}, v\right) \xrightarrow{a} \left(\vec{l'}, v'\right)$, the following conditions have to be satisfied: the transition has to be a valid transition in terms of definition for network of automata (see Definition 4), upon exercise of transition the values of clocks have to satisfy restrictions associated with them, appropriate clocks associated with active transitions have to be reset, and finally all the invariants assigned to all the final locations have to be satisfied. The delay transition describes a transition in time. In this case, all the active locations stay unchanged and only the values of the clocks are modified. Thus all invariants have to be correct during this modification.

A typical formal definition of semantics of network of timed automata is presented below. For any synchronization $t = (t_1, \ldots, t_n) \in Sync$, $I_s =$

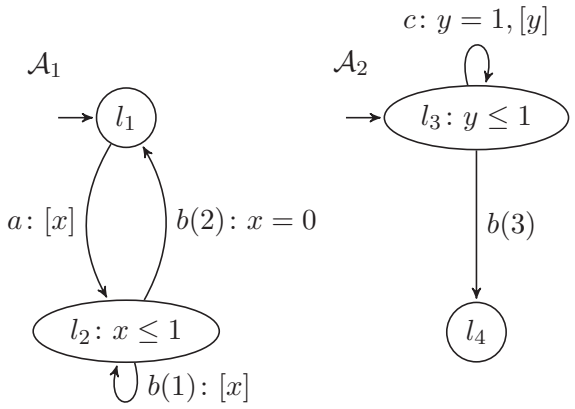


Figure 2.3: Network of two timed automata

$\{i \in [1..n] \,|\, t_i \neq \bullet\}$ denotes the set of indices of automata concerned by the transition.

Definition 9. ***(Semantics of NTA)*** The semantics of network of timed automata $\mathcal{N} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ can be defined by the following timed transition system $(Q, q_0, \Sigma, \rightarrow)$ where: DEF

- $Q = \left\{ \left(\vec{l}, v\right) \in (L_1 \times \ldots \times L_n) \times (X \rightarrow \mathbb{R}_{\geq 0}) \,\middle|\, v \models \bigwedge_i Inv_i\left(l_i\right) \right\}$,
- $q_0 = \left(\vec{l_0}, v_0\right)$, such that $\forall x \in X, v\left(x\right) = 0$,
- the transition relation $\rightarrow \in Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$ is composed of:
  - action transition: $\left(\vec{l}, v\right) \xrightarrow{a} \left(\vec{l'}, v'\right)$ iff
    - $\exists q = (t_1, \ldots, t_n) \in Sync\,, \forall i \leq n\,,$ $\begin{cases} l'_i = l_i \wedge t_i = \bullet & \text{if } a \notin \Sigma_i \\ t_i = (l_i, g_i, a, r_i, l'_i) & \text{otherwise} \end{cases}$
    - $v \models \bigwedge_{i \in I_S} g_i$, $v' = v\left[\bigcup_{i \in I_S} r_i\right]$, and $v' \models \bigwedge_i Inv_i\left(l'_i\right)$
  - delay action: $\forall d \in \mathbb{R}_{\geq 0}, \left(\vec{l}, v\right) \xrightarrow{d} \left(\vec{l}, v + d\right)$ iff $\forall d' \in [0, d]\,, v + d' \models \bigwedge_i Inv_i\left(l_i\right)$. ♦

Example 4. Let us consider an example in Figure 2.3 which is an extension of our previous example in Figure 2.1b. There are two timed automata with two clocks x and y. Each of the locations is marked with a label and invariants. The transitions are labeled with their action (a, b, or c), the guard on clocks, and the set of resets (in square brackets). Let us consider a EXM

simple scenario: $((l_1, l_3), (0,0)) \xrightarrow{1} ((l_1, l_3), (1,1)) \xrightarrow{c} ((l_1, l_3), (1,0)) \xrightarrow{a} ((l_2, l_3), (0,0)) \xrightarrow{0.5} ((l_2, l_3), (0.5, 0.5)) \xrightarrow{b} ((l_2, l_4), (0, 0.5))$. As we can observe in the beginning of our scenario, the first executed transition is an action transition c. In order to fire this transition, we have to wait one unit ($y = 1$). Than we reset the clock y. After that, we decide to execute transition a which causes reset of clock x. Half a unit of time later, we execute the synchronization b which resets the clock x. What is interesting to note: having only the scenario, we can deduce which specific transitions were fired. Namely, if we ignore for a moment all the guards, for the state $((l_2, l_3), (0.5, 0.5))$, there are two possible synchronizations with the same label b. But since we know that the values of both clocks are 0.5 and that there is a guard $x = 0$ next to the transition $b(2)$, we know that only the synchronization of $b(1)$ with $b(3)$ is possible. This problem will be clearer once we discuss the problem of supervision.

2.2.5 Network of parametric automata with linear constraints

In the further part of the book we use also a network of parametric automata with linear constraints (see Section 5.4). This type of model is an extension of timed automata and is much more expressive. The main difference consists of the form of constraints assigned to locations and transitions of automata. Each constraint can represent a not necessarily closed convex polyhedron (NNC Polyhedron; see *e.g.* [9]). In other words, it is a conjunction of linear constraints (strict or non-strict) (see also Section 5.4.1) which represent a finite number of open or closed affine half-spaces.

Let $\mathcal{C}_P(Y)$ be a system of linear constraints on Y (see the beginning of the chapter).

DEF **Definition 10. (Parametric automaton with linear constraints)** A parametric automaton with linear constraints is a tuple $\langle L, l_0, X, \Sigma, T, Inv_\Pi, \Pi, D_\Pi \rangle$ where:

- L is a finite set of locations,
- $l_0 \in L$ is the initial state,
- X is a finite set of clocks,
- Σ is a finite alphabet of actions,
- $T \subseteq L \times \mathcal{C}_P(X \cup \Pi) \times \Sigma \times 2^X \times L$ is a finite set of transitions,
- $Inv_\Pi : L \to \mathcal{C}_P(X \cup \Pi)$ assigns an invariant to any location.
- Π is a finite set of parameters and $\Pi \cap (L \cup T \cup X) = \emptyset$, and
- D_Π is a conjunction of linear constraints describing the set of *initial constraints* on the parameters. ♦

Given a notion of a single parametric automata we define a network of automata.

Definition 11. (Network of parametric automata with linear constraints) A network of parametric automata with linear constraints (NPA) is a set $\mathcal{N} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ where $\mathcal{A}_i = \langle L_i, l_{0i}, X_i, \Sigma_i, T_i, Inv_{\Pi_i}, \Pi_i, D_{\Pi_i} \rangle$ is a parametric automata with linear constraints. We make the assumption that **clocks are not shared** between automata. DEF

We assume that there exists a set of labels describing synchronizations Σ_{sync}. There also exists a function $sync : \bigcup T_i \to 2^{\Sigma_{sync}}$. The activity of the automata is synchronized on transitions having the same synchronization labels. ♦

We can note that the method of synchronization is slightly different in comparison with a network of timed automata from Definition 4. This method of synchronization reflects, for example, the mechanism used in Spinta (see Section 5.4).

The semantics of an NPA is quite similar to the one of a network of timed automata. In order to define it, we assume below that we already know the semantics of a network of automata with linear constraints and without parameters. This assumption comes from the fact that the only difference between networks of timed automata and networks of automata with linear constraints and without parameters is the form of expressions used in invariants and guards.

Definition 12. ***(Semantics of NPA)*** Given an NPA $\mathcal{N} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ in which $\mathcal{A}_i = \langle L_i, l_{0i}, X_i, \Sigma_i, T_i, Inv_{\Pi_i}, \Pi_i, D_{\Pi_i} \rangle$, its semantic can be defined by a valuation $v \in \bigcap_{i=1}^{n} D_{\Pi_i}$ and the semantics of *a network of automata with linear constraints (without parameters)* $\mathcal{N}_v = \{\mathcal{A}'_1, \ldots, \mathcal{A}'_n\}$ in which $\mathcal{A}'_i = \langle L_i, l_{0i}, X_i, \Sigma_i, T_i, Inv_i \rangle$, such that DEF

$$\forall l \in \bigcup_{i=1}^{n} L_i \,,\; Inv_i\,(l) = Inv_{\Pi_i}\,(l)\,(v)$$

$$\forall t \in \bigcup_{i=1}^{n} T_i \,,\; \gamma_i\,(t) = \gamma_{\Pi_i}\,(t)\,(v)$$

where $\gamma_{\Pi_i}\,(t)$ denotes a guard of the transition t in the automaton $\mathcal{A}_i$, $\gamma_i\,(t)$ denotes a guard of the transition t in the automaton $\mathcal{A}'_i$. ♦

2.3 Time Petri nets

In this part of the chapter, we present the second class of models which will be dealt with later in the book, namely, time Petri nets. Similarly to the previous section, presentations is divided into several parts. At the

beginning of the presentation we give a general outline of the safe Petri nets and key concepts related to them. Then, we move onto the aspect of time constraints in Petri nets, namely, we define so called time Petri nets. The last part presents a generalized version of the time Petri net, *i.e.* a parametric time Petri net which potentially gives us more possibilities of application.

2.3.1 Safe Petri nets

Petri nets proposed by Petri in [78], like networks of timed automata, are used for modeling distributed systems. However, the structure and mode of operation offered by Petri nets is significantly different from networks of timed automata. Our introduction to Petri nets starts with a definition of a place/transition net to which we will often refer in the rest of the work (see *e.g.* Section 2.6.1) as it is a kind of framework for more complex structures including Petri nets.

DEF **Definition 13.** ***(Place/transition net)*** A *place/transition net* (P/T net) is a tuple $\langle P, T, W \rangle$ where:

- P is a finite set of *places*,
- T is a finite set of *transitions*, with $P \cap T = \emptyset$, and
- $W \subseteq (P \times T) \cup (T \times P)$ is the *transition incidence relation.*

This structure defines a directed bipartite graph such that $(x, y) \in W$ iff there is an arc from x to y. We further define, for all $x \in P \cup T$, the following sets:

- ${}^\bullet x = \{y \in P \cup T \,|\, (y, x) \in W\}$, and
- $x^\bullet = \{y \in P \cup T \,|\, (x, y) \in W\}$.

These set definitions naturally extend by union to subsets of $P \cup T$, *i.e.* given a set $X \subseteq P \cup T$, ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$ and $X^\bullet = \bigcup_{x \in X} x^\bullet$. ♦

A *marking* M is a mapping in $\mathbb{N}^P$ such that, for each $M \in \mathbb{N}^P$, $M(p_i)$ denotes the number of tokens in the place p_i. Having the notion of marking, below we formalize what Petri nets are.

DEF **Definition 14.** ***(Petri net)*** A *Petri net (PN)* is a marked P/T net, *i.e.* a pair $(\mathcal{N}, M_0)$ where $\mathcal{N} = \langle P, T, W \rangle$ is a P/T net and $M_0 \in \mathbb{N}^P$ is a marking of $\mathcal{N}$, called the initial marking. ♦

A transition t of a Petri net may be executed iff there is a token at each of its input place. When the transition fires, it consumes all the mentioned tokens and creates one token in each output place.

In this document we restrict our study to *1-safe* nets, *i.e.* nets such that for each $p \in P$, $M(p) \leq 1$. Therefore, in the rest of the work, we will usually

identify the marking M with the set of places in which for each place p, $M(p) = 1$. Note that k-safe Petri nets ($M(p) \leq k$) can be reduced to 1-safe Petri net (see [23]). A transition $t \in T$ is enabled in a marking M iff $M \geq {}^\bullet t$. Moreover we denote by $\mathsf{enabled}(M)$ the set of transitions enabled by m. A transition t' is *newly enabled* after executing t from the marking M iff it is not enabled by $M - {}^\bullet t$ and it is enabled by $M - {}^\bullet t + t^\bullet$. Formally, it can be expressed as follows:

$$\uparrow enabled(t', M, t) = \left(M - {}^\bullet t + t^\bullet \geq {}^\bullet t'\right) \wedge \left[\left(M - {}^\bullet t < {}^\bullet t'\right) \vee \left(t = t'\right)\right]$$

2.3.2 Time Petri nets

Time Petri nets were described for the first time in [72]. This model was introduced as an extended version of Petri nets. It is additionally equipped with time constraints imposed on starting times for transitions. As we will see below, these constraints take the form of time intervals assigned to each transition. If clock values for a given transition are within its time interval, the transition can be executed. It is worth noting that the model should not be confused with the timed Petri nets [79] in which transitions are fired as soon as they are enabled.

Definition 15. ***(Time Petri net)*** A *time Petri net (TPN)* is a tuple $\langle P, T, W, M_0, \mathsf{eft}, \mathsf{lft}\rangle$ where: DEF

- $\langle P, T, W, M_0\rangle$ is a Petri net and
- $\mathsf{eft} : T \to \mathbb{Q}_{\geq 0}$ and $\mathsf{lft} : T \to \mathbb{Q}_{\geq 0} \cup \{\infty\}$ associate the earliest firing delay $\mathsf{eft}(t)$ and the latest firing delay $\mathsf{lft}(t)$ with each transition t. ♦

A state of time Petri net is a pair (M, v) where M is a marking and $v \in (\mathbb{Q}_{\geq 0})^T$ is a valuation such that each value v_i is the elapsed time since transition t_i was last enabled.

Definition 16. ***(Semantics of time Petri net)*** The semantics of a time Petri net $\langle P, T, F, M_0, \mathsf{eft}, \mathsf{lft}\rangle$ is a timed transition system $\langle Q, q_0, \Sigma, \to\rangle$ where: DEF

- $Q = \mathbb{N}^P \times (\mathbb{Q}_{\geq 0})^T$,
- $q_0 = \left(M_0, \vec{0}\right)$,
- the transition relation $\to \in Q \times (T \cup \mathbb{Q}_{\geq 0}) \times Q$ consists of two transition relations:
  - the discrete transitions are defined for all $t_i \in T$ by

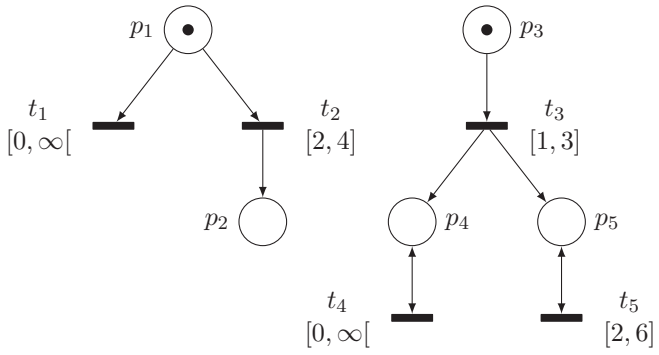


Figure 2.4: A time Petri net

$$(M,v) \xrightarrow{t_i} \left(M', v'\right) \text{ iff } \begin{cases} M \geq {}^\bullet t_i \wedge M' = M - {}^\bullet t_i + t_i{}^\bullet \\ \mathsf{eft}\left(t_i\right) \leq v_i \leq \mathsf{lft}\left(t_i\right) \\ v'_k = \begin{cases} 0 & \text{if } \uparrow \mathit{enabled}\left(t_k, M, t_i\right) \\ v_k & \text{otherwise} \end{cases} \end{cases}$$

– the continuous transition are defined for all $d \in \mathbb{Q}_{\geq 0}$ by

$$(M,v) \xrightarrow{d} \left(M, v'\right) \text{ iff } \begin{cases} v' = v + d \\ \forall t_k \in \mathsf{enabled}\left(M\right) \,,\, v'_k \leq \mathsf{lft}\left(t_k\right) \end{cases}$$

♦

A run of a time Petri net is a path starting in q_0 and for which, like in timed transition systems, we can consider alternation of two types of operations: discrete and continuous ones.

EXM

Example 5. In Figure 2.4, we can observe an example of a time Petri net which consists of two separate components. The initial marking is $\{p_1, p_3\}$. There are five transitions, each of which has a certain time interval assigned to it. The initial marking enables three transitions. However, when we consider their time constraints and the initial state of the model, denoted by q_0, we note that only the transition t_1 is available at the global time 0. Now let us shortly analyze the following scenario of the system.

- In the beginning, we decide to wait two units of time without moving the two tokens.
  $q_0 \xrightarrow{2} q_1$, $M_1 = M_0$, $v\left(t_1\right) = v\left(t_2\right) = v\left(t_3\right) = 2$;

- After two units of time, it appears that not only t_1 but also t_2 and t_3 are available for execution. We can nondeterministically chose to fire one of the transitions. We decide to execute t_3. We can observe that one token was consumed and two new tokens were produced. This fact implies that time for transitions related to places p_4 and p_5 starts to progress.

  $q_1 \xrightarrow{t_3} q_2,\ M_2 = \{p_1, p_4, p_5\},\ v\,(t_1) = v\,(t_2) = 2, v\,(t_4) = v\,(t_5) = 0;$

- We decide to spend two units of time after which we can observe that the transition t_2 becomes urgent and has to be fired immediately.

  $q_2 \xrightarrow{2} q_3,\ M_3 = M_2,\ v\,(t_1) = v\,(t_2) = 4, v\,(t_4) = v\,(t_5) = 2;$

- We fire the transition t_2.

  $q_3 \xrightarrow{t_2} q_4,\ M_4 = \{p_2, p_4, p_5\},\ v\,(t_4) = v\,(t_5) = 2;$

- Three units of time elapse.

  $q_4 \xrightarrow{3} q_5,\ M_5 = M_4,\ v\,(t_4) = v\,(t_4) = 5;$

- Finally, we fire the transition t_5 which represents a loop and does not change the marking.

  $q_5 \xrightarrow{t_5} q_6,\ M_6 = M_5\ v\,(t_2) = v\,(t_3) = 4, v\,(t_4) = 1;$

In the example, we can observe important properties of time Petri nets such as: notion of urgency, influence of time constraints onto availability of transitions, nondeterminism, loops. In Figure 2.4 we can observe one more transition which is not executed in the scenario, *i.e.* t_4. We mention it because of its importance in terms of Chapter 4 as it represents a special kind of loops. In theory, the transition t_4 is a potential source of what is called *Zeno behavior*, *i.e.* it can be fired infinitely many times in a certain finite amount of time. The presence of such loops in the model can simply be verified by detection of loops with the minimal execution time equal to 0. It is not always possible to remove such behaviors from models. That is why we have to deal somehow with them depending on the problem we consider.

2.3.3 Parametric time Petri nets

A mainstream way of adding time to Petri nets is by equipping transitions with a time interval [72, 20] as we described in the previous section. In this section, we consider an extension allowing the designer to leave open the knowledge of time bounds by putting symbolic expressions on parameters in time intervals instead of rational constants.

Definition 17. ***(Parametric time Petri net)*** A *parametric time Petri net (PTPN)* is a tuple $\langle P, T, W, M_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$ where: DEF

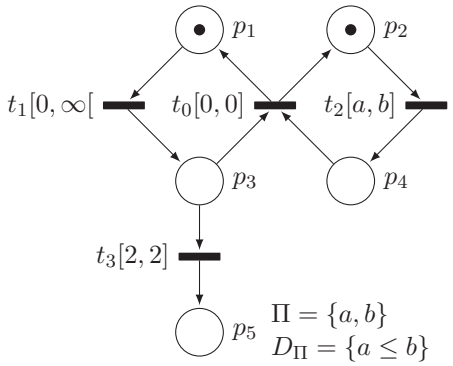


Figure 2.5: A parametric Petri net

- $\langle P, T, W, M_0 \rangle$ is a Petri net,
- Π is a finite set of *parameters* and $\Pi \cap (P \cup T) = \emptyset$,
- D_Π is a conjunction of linear constraints describing the set of *initial constraints* on the parameters, and
- $\mathsf{eft} : T \to \mathbb{Q}_{\geq 0} \cup Expr(\Pi)$ and $\mathsf{lft} : T \to \mathbb{Q}_{\geq 0} \cup \{\infty\} \cup Expr(\Pi)$ are functions respectively called *earliest (*eft*)* and *latest (*lft*)* transition firing times. For each transition $t \in T$, if $\mathsf{eft}(t)$ and $\mathsf{lft}(t)$ are constants, it is assumed that $\mathsf{eft}(t) \leq \mathsf{lft}(t)$, otherwise, it is assumed that $D_\Pi \Rightarrow \mathsf{eft}(t) \leq \mathsf{lft}(t)$. ⧫

DEF **Definition 18.** ***(Semantics of parametric time Petri net)*** Given a parametric time Petri net $\mathcal{N} = \langle P, T, W, M_0, \mathsf{eft}_\Pi, \mathsf{lft}_\Pi, \Pi, D_\Pi \rangle$ its semantic can be defined by a valuation $v \in D_\Pi$ and semantics of time Petri net $\mathcal{N}_v = \langle P, T, W, M_0, \mathsf{eft}, \mathsf{lft} \rangle$ such that

$$\forall t \in T\,, \begin{cases} \mathsf{eft}(t) = \mathsf{eft}_\Pi(t)(v) \\ \mathsf{lft}(t) = \mathsf{lft}_\Pi(t)(v) \end{cases}$$

⧫

EXM **Example 6.** Figure 2.5 gives an example of a parametric time Petri net. Notice that the time interval of transition t_2 refers to two parameters a and b. The only initial constraint is $D_\Pi = \{a \leq b\}$.

2.4 Decidability and other interesting issues

Networks of timed automata

A network of timed automata can be seen as the parallel composition $\mathcal{A}_1 \mid \cdots \mid \mathcal{A}_n$ of a set of timed automata. As such, it does not add expressive power in relation to a timed automata. Therefore, it is usually sufficient to consider only a single timed automaton in order to prove decidability of various problems.

The technique which was used to prove decidability of many verification problems for timed automata was directly introduced by the authors of the model ([5, 6]). The technique was based on the construction of a special region automaton. This method was initially used to prove the reachability of a control state in a timed automaton. The principle of this technique relies on an abstraction of the behavior of the timed automaton. Namely, the authors used the fact that constant values in the time constraints of the model are countable. Thus, following this fact, they decided to group some states for which the trajectories of the model are always the same. This let them create a so called region graph in which each region represents a group of the similar states. Finally, from the region graph, the related finite region automaton was constructed. This way, it was proven in [6] that checking the reachibility of a location in a timed automaton is decidable and that it is a PSPACE-Complete problem.

It appeared that, using this technique, several verification problems such as untimed language inclusion or language emptiness can be solved. Unfortunately, the language inclusion problem and the universality problem are undecidable [5, 6]. Moreover, timed automata can not be complemented, *i.e.* there exists a timed language whose complement cannot be accepted by any timed automaton. Untimed bisimilarity for timed automata is decidable in EXPTIME [63].

It is worth mentioning that in order to improve verification of timed automata, a more efficient technique was later developed. The technique is based on so called *zones.* Intuitively, a zone represents a set which groups all regions satisfying some time constraints. It is well-known that such sets can be efficiently represented and stored in memory as DBMs (Difference Bound Matrices) [13]. For example, this technique is used in a software tool called Uppaal [64, 14].

Timed automata with diagonal clock constraints

In Definition 6 of timed automata, the only possible form of timed constraints was $x \bowtie c$. However, already in the original work of Alur and Dill, a different type of constraints was mentioned. Namely, the diagonal constraints of the form $x - y \bowtie c$ in which x and y stand for clocks and c is an integer. In terms of modeling, the diagonal constraints give us a better way to express

time constraints which are more difficult and more complex to write using the constraints $x \bowtie c$. This was already shown for example in [24] that application of diagonal constraints give a more compact form of the whole model.

In the context of expressiveness a timed automaton with diagonal constraints has the same power as the original timed automata with simple constraints. This was proven in [19]. The main principle of the translation between the two versions of the model is as follows: for each diagonal constraint $x - y \bowtie c$, two copies of the original automata are made. In one of the copies, the constraint $x - y \leq c$ is satisfied; and in the second one it is $x - y > c$ which is hold. Depending on the values of clocks (which depend on reset to zero), we use the appropriate copy of the automaton. As we can expect, the translation procedure suffers from an exponential blowup in the number of diagonal constraints.

It is worth noting that checking reachability of such timed automata is decidable.

Apart from the diagonal constraints, there are also some other types of time constraints which were studied, like for example: additive time constraints which are of the form $x + y \bowtie c$. It is more expressive than the original timed automata. Unfortunately, in the case of additive time constraints, the problem of reachability is in general undecidable.

Updates of clocks

Another extension of the original timed automaton is based on different types of clock update operations. In the basic model, the only possibly modification of the value of the clocks is the reset to zero ($x := 0$). However, an update operation can have many other forms. For example, instead of reseting a clock to zero, the clock can be reset to some constant c. Thus, we obtain an update operation $x := c$. We can even extend this operation and use more than one clock, e.g. $x := y + c$, where x, y are clocks.

Updatable timed automata, as automata with update operations are called, were analyzed for example in [25]. It appeared that many of the considered update operations were too powerful and made the reachability problem undecidable. However, it has been proved that the reachability problem for timed automata with updates of the form $x := c$ is decidable.

Some extensions and subclasses

Linear hybrid automata ([80]) are not of a special interest in our book. However, it is still one of the well-known extensions of timed automata. When we compare it to the original timed automata, we can note differences like: general linear functions in constraints, updates of derivatives of variables, more general updates on variables. Unfortunately, many verification problems for

linear hybrid automata are undecidable.

Since even the original version of timed automaton was found too hard for model verification, some simpler subclasses of the model were also investigated.

One of the possible simplifications of the model is decrement of the number of clocks. The first case which was of special interest was a timed automaton with one clock. It was found that many verification problems can be checked quite efficiently for this type of automaton. Besides, problems like timed language inclusion for finite words, emptiness for one-clock alternating automata are decidable ([76, 65]).

Another known subclass which was introduced in [7] is called event-recording automata. This particular class of timed automata assigns to each action a corresponding clock. Thus, the set of clocks is $X = \{x_a \mid a \in \Sigma\}$.

Time Petri nets

One of the key problems for time Petri nets is reachability of a given marking and boundedness. Both these issues have been addressed in [59], in which the authors proved that they are undecidable. The consequence of this is also the undecidability of problems such as liveness and reachability of states.

Similarly to the case of networks of timed automata, in order to increase the efficiency of verification of time Petri nets, abstraction methods were proposed. These methods rely on the grouping of certain states creating this way classes so that the reachability analysis may be conducted on them. Examples of this approach can be found in the work of [22, 20] which proposed the *state class graph*, and in [50] where the authors describe the *zone graph*.

Some extensions

The main extension of time Petri net we are interested in our work is a parametric time Petri net. We have already mentioned them in this chapter when presenting each of the models. As we will see in the next section, such a model can *e.g.* control the behavior of the model by changing parameters in the intervals associated with transitions. This model has been described in [86].

Another interesting extension of a time Petri net is a *stopwatch Petri net* described in [21]. This model allows for suspension and resumption of actions.

The definition of this model is additionally extended by *stopwatch incidence relation*, $W_s \subseteq P \times T$. Intuitively, when we look into the semantics of time Petri net, it states that any enabled transition measures the time during which it has been enabled and an enabled transition can only fire if that time is within the time interval of the transition. Also, unless it is disabled by the firing of another transition, the transition must fire within

the interval: a finite upper bound for the time interval then means that the transition will become urgent at some point. For stopwatch Petri net, the time during which the transition t has been enabled progresses if and only if all its *activating* places, that are places in $\{p \in P \mid (p,t) \in W_s\}$, are marked. Otherwise it is "frozen" and keeps its current value.

2.5 Comparison of timed automata and time Petri nets

Knowledge about similarities and differences of the two models can be crucial in the design process of distributed systems. It not only allows the designer to choose the right model as the basis for the system, but it also helps to improve, modify, and exploit in some new ways already existing systems.

So far there has been many works conducted in order to describe both networks of timed automata and time Petri nets. They have proven its usefulness in many practical, real life problems. There is a significant number of various applications in many outwardly different domains. However, many of these problems have been specifically addressed for certain subclasses of the models. Therefore, the desire to compare the two models appears to be the most natural and desirable. Translation between the models opens a great number of possibilities to reuse many solutions which, for the moment, are available for only one of the models. We can imagine that there is a specific algorithm to check a property on Petri nets and that does not exist for timed automata. If we can translate the timed automata into Petri net we would probably be able to check the property for timed automata without inventing and developing new methods and algorithms. This approach was already adapted in many cases.

In this section, we present the most popular ways of comparing models with time constraints. Then we briefly present the existing results concerning the conversion of networks of timed automata into time Petri nets, and vice versa.

2.5.1 Timed similarity

There are many ways in which systems with time can be compared. In this section, we will focus on the most interesting ones which are *timed language equivalence*, *strong timed similarity, weak timed similarity,* and *distributed timed language equivalence.*

We start with the most basic class of equivalence which is the equivalence of timed languages. Below, we use notions introduced in Section 2.1.

DEF **Definition 19.** ***(Equivalence of timed languages)*** Let $\mathcal{T}_1 = \langle S_1, S_0^1, \rightarrow_1, \Sigma_1 \rangle$ and $\mathcal{T}_2 = \langle S_2, S_0^2, \rightarrow_2, \Sigma_2 \rangle$ be two timed transition systems. $\mathcal{T}_1$ and $\mathcal{T}_2$

are language equivalent if $\mathcal{L}(S_1) = \mathcal{L}(S_2)$ for acceptance conditions defined as usual from final states or Büchi conditions. ♦

Below we present two different timed similarity relations which are sometimes called *branching equivalences*. This is caused by the fact that they take into account the branching structures of the concerned timed transition systems.

Definition 20. ***(Strong timed similarity)*** Let $\mathcal{T}_1 = \langle S_1, S_0^1, \rightarrow_1, \Sigma_1 \rangle$ and $\mathcal{T}_2 = \langle S_2, S_0^2, \rightarrow_2, \Sigma_2 \rangle$ be two timed transition systems. Let $\preceq$ be a binary relation over $S_1 \times S_2$. We say that $\preceq$ is a strong timed simulation relation of $\mathcal{T}_1$ by $\mathcal{T}_2$: DEF

1. if $s_1 \in S_0^1$ there exists some $s_2 \in S_0^2$ such that $s_1 \preceq s_2$,
2. if $s_1 \xrightarrow{d}_1 s_1'$ with $d \in \mathbb{R}_{\geq 0}$ and $s_1 \preceq s_2$, then $s_2 \xrightarrow{d}_2 s_2'$ for some s_2', and $s_1' \preceq s_2'$,
3. if $s_1 \xrightarrow{a}_1 s_1'$ with $a \in A$ and $s_1 \preceq s_2$, then $s_2 \xrightarrow{a}_2 s_2'$ and $s_1' \preceq s_2'$.

If there exists a strong timed simulation relation of $\mathcal{T}_1$ by $\mathcal{T}_2$, we say that $\mathcal{T}_2$ strongly simulates $\mathcal{T}_1$ and we denote it by $\mathcal{T}_1 \preceq_{\mathcal{S}} \mathcal{T}_2$. ♦

When there are two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$, and there exists a strong simulation relation $\preceq_{\mathcal{S}}$ such that $\mathcal{T}_1 \preceq_{\mathcal{S}} \mathcal{T}_2$ and $\mathcal{T}_2 \preceq_{\mathcal{S}}^{-1} \mathcal{T}_1$, where $\preceq_{\mathcal{S}}^{-1} \equiv \succeq_{\mathcal{S}}$, we say that $\preceq_{\mathcal{S}}$ is a strong timed bisimulation. To emphasize this situation, we write $\mathcal{T}_1 \approx_{\mathcal{S}} \mathcal{T}_2$ instead of $\mathcal{T}_1 \preceq_{\mathcal{S}} \mathcal{T}_2$. Moreover, the existence of such a relation between the two systems means that they are strongly timed bisimilar.

Below, we define another type of similarity which is called *weak similarity*. Unlike in the case of strong similarity, weak similarity gives the possibility of simulating a single move by a sequence. It is also stronger than language equivalence and it is one of the most common equivalence relations for timed systems.

Before we introduce the notion of weak similarity, we define a modified version of a timed transition system $\mathcal{T} = \langle S, S_0, \rightarrow, \Sigma_\epsilon \rangle$ (see *e.g.* [16]) by removing ϵ transitions . Thus $\mathcal{T}^\epsilon = \langle S, S_0^\epsilon, \rightarrow_\epsilon, \Sigma \rangle$ is defined as follows:

- $s \xrightarrow{d}_\epsilon s'$, where $d \in \mathbb{R}_{\geq 0}$ iff there exists a run $\rho = s \xrightarrow{*} s'$ such that $Untimed(\rho) = \epsilon$ and $Duration(\rho) = d$,
- $s \xrightarrow{a}_\epsilon s'$, where $a \in \Sigma$ iff there exists $\rho = s \xrightarrow{*} s'$ such that $Untimed(\rho) = a$ and $Duration(\rho) = 0$,
- $S_0^\epsilon = \Big\{ s \in S \mid \exists s' \in S_0 \mid s' \xrightarrow{*} s \equiv \rho \wedge Duration(\rho) = 0 \wedge Untimed(\rho) = \epsilon \Big\}$.

DEF **Definition 21.** ***(Weak timed similarity)*** Let $\mathcal{T}_1 = \langle S_1, s_0^1, \rightarrow_1, \Sigma_1 \rangle$ and $\mathcal{T}_2 = \langle S_2, s_0^2, \rightarrow_2, \Sigma_2 \rangle$ be two timed transition systems. Let $\preceq$ be a binary relation over $S_1 \times S_2$. $\preceq$ is a *weak timed simulation* relation of $\mathcal{T}_1$ by $\mathcal{T}_2$ if it is a strong timed simulation relation of $\mathcal{T}_1^{\varepsilon}$ by $\mathcal{T}_2^{\varepsilon}$. If there exists a weak timed simulation relation of $\mathcal{T}_1$ by $\mathcal{T}_2$, we say that $\mathcal{T}_2$ weakly simulates $\mathcal{T}_1$ and we denote it by $\mathcal{T}_1 \preceq_{\mathcal{W}} \mathcal{T}_2$. ♦

When there are two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$, and there exists a weak simulation relation $\preceq_{\mathcal{W}}$ such that $\mathcal{T}_1 \preceq_{\mathcal{W}} \mathcal{T}_2$ and $\mathcal{T}_2 \preceq_{\mathcal{W}}^{-1} \mathcal{T}_1$, where $\preceq_{\mathcal{W}}^{-1} \equiv \succeq_{\mathcal{W}}$, we say that $\preceq_{\mathcal{W}}$ is a weak timed bisimulation. To emphasize this situation, we write $\mathcal{T}_1 \approx_{\mathcal{W}} \mathcal{T}_2$ instead of $\mathcal{T}_1 \preceq_{\mathcal{W}} \mathcal{T}_2$. Moreover, existence of such a relation between the two systems means that they are weakly timed bisimilar.

Note that, for the three presented relations, we have the following dependencies:

- if $\mathcal{T}_1 \preceq_{\mathcal{S}} \mathcal{T}_2$ then $\mathcal{T}_1 \preceq_{\mathcal{W}} \mathcal{T}_2$, and
- if $\mathcal{T}_1 \preceq_{\mathcal{W}} \mathcal{T}_2$ then $\mathcal{L}(\mathcal{T}_1) \subseteq \mathcal{L}(\mathcal{T}_2)$.

The last type of equivalence relation we present is based on timed traces which can be used to represent a partial order of executions of real-time distributed systems. It is especially interesting when comparing distribution of actions. Below we use a notion of process which, in short, represents all elements of activity of the system, *i.e.* mainly events and order between them.

DEF **Definition 22.** ***(Timed trace, [11])*** A *timed trace* over the alphabet Σ, and the set of processes $\Pi = (\pi_1, \ldots, \pi_n)$ is a tuple $\mathcal{W} = (E, \preceq, \lambda, t, proc)$ where:

- E is a set of events,
- $\preceq \subseteq E \times E$ is a partial order,
- $\lambda : E \rightarrow \Sigma$ is a labeling function,
- $t : E \rightarrow \mathbb{R}_{\geq 0}$ assigns a date to each event. Moreover, if $e_1 \preceq e_2$, then $t(e_1) \leq t(e_2)$,
- $proc : \Sigma \rightarrow 2^{\Pi}$ maps each action to a subset of Π. ♦

The *distributed timed language* is a set of timed traces.

EXM **Example 7.** In Figure 2.6, we consider a simple example of a timed trace with five events. As we can see, each event is characterized by a pair $(\lambda(e), t(e))$. A possible word for the timed trace is $(a,1)(b,2)(c,3)(e,4)(d,4)$.

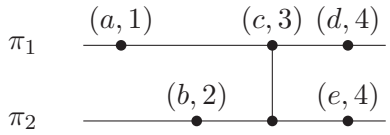


Figure 2.6: A timed trace

Definition 23. ***(Equivalence of distributed timed languages)*** If two timed transition systems produce the same set of timed traces, we say that they accept the same distributed timed languages. ♦

DEF

Note that if two timed transition systems have the same distributed timed languages they are timed bisimilar.

2.5.2 Translation between the models

The works which compare the expressiveness of both models are relatively new when we compare them with the time of their introduction. Below, we review some key results on the comparison of time Petri nets and networks of timed automata.

One of the first interesting results was described in [56]. It was proven that subclass of timed automata called *timed state machines* (diagonal-free, without invariants, with strict constraints) is weakly timed bisimilar to 1-safe non-Zeno time Petri nets with non-urgent semantics. A more general translation was presented in [66, 67] where it was shown that the state class graph construction of a bounded TPN can be represented as a timed automaton. The automaton is called state class automaton. The timed automaton and TPN are weakly timed bisimilar. However, in terms of complexity, the construction appeared to be expensive and impractical.

On the other side, in [16] it was shown that there exists a TA $\mathcal{A}$ such that there is no TPN that is weakly *timed similar* to $\mathcal{A}$. In other words, TAs are strictly more expressive than bounded TPNs with respect to weak timed bisimilarity. The authors propose a structural translation from TA to 1-safe TPN that preserves timed language acceptance.

Later, in [31, 30], another structural translation from TPN to TA was presented that preserves weak timed bisimilarity of the TPN. Moreover, the translation was implemented in Romeo [50]. This opened a possibility to use existing software, called UPPAAL, which was originally designed to verify networks of timed automata. In general, structural translation was found to be a good alternative to the previous methods. The translation is syntactic and it does not need expensive computations like in the case of the state class graphs. Moreover, there is an easy correspondence of the clocks of the TA with the timing constraints on the transitions of the base TPN. It was also shown in [31] that the bounded TPN is a subclass of the class of TA with respect to weak timed bisimilarity. Hence, for each safe time Petri net

$\mathcal{N}$, there exists a TA which is weakly timed bisimilar to $\mathcal{N}$.

Another approach to the problem of translation was presented in [38]. The authors translate safe TPN to TA with avoiding the timed reachability step. The method is based on the underlying untimed Petri net. They construct what is called marking class timed automata. The translation is strong timed bisimilar.

Knowing that TPN is strictly included in TA and that there exist TAs that are not weakly timed similar to any TPN, the authors of [17] describe a subclass of the class TA such that, for each automaton from that subclass, we can find a TPN which is weakly timed bisimilar. For this purpose the authors define uniform bisimilarity which is stronger than weak timed bisimilarity. They show that the problem of deciding whether there is a TPN bisimilar to TA is PSPACE-complete.

Quite recently, in [11], it was noticed that the translation procedures which were introduced so far do not keep concurrency relations. Thus, they decided to propose a translation technique from TPN to NTA which preserves distributed timed language. In this approach, the untimed version of the given time Petri net is decomposed into a number of subnets. Then, each of the subnets is translated into an automaton with one clock. Finally, the time constraints and synchronizations are translated. However, in order to cope with the time constraints and the synchronizations, the authors introduced special global invariants into the final NTA.

If the reader is interested in more information on the comparison of the two models, we refer to [82]. The author compares three models: TA, NTA and additionally TAPN (timed-arc Petri net) and describes weak and strong sides of all the models, their expressiveness and possibilities of translation.

2.6 Observations and explanations

In the next chapter, we closely study topic of monitoring in distributed systems. Before, however, we look at two basic elements that are part of the monitoring process, *i.e.*: *observations* and *explanations*. Earlier, we also introduce the semantics of true concurrency which is essential for structures such as unfoldings of network of automata or unfoldings of Petri nets.

2.6.1 Semantics of true concurrency

The main task of the formal models which have been described is modeling of distributed systems. However, to fully reflect the nature of these systems and to capture all interesting properties, we have to consider more than the standard sequential semantics. For this purpose, below we introduce some of the key notions for the structures that will enable us a precise definition of causal relations between events, for example, parallel events. A more detailed

description of these structures taking into account the time constraints is presented in the next Chapter 3.

First, let us recall the notion of P/T net (see Definition 13) and describe it in more details.

We say that there is a path $x_1, x_2, \ldots, x_n$ in a P/T net iff $\forall i \in [1..n]$, $x_i \in P \cup T$ and $\forall i \in [1..n-1]$, $(x_i, x_{i+1}) \in W$.

In an *acyclic* P/T net, let us consider two distinct elements $x, y \in P \cup T$.

- x and y are *causally related*, which we denote by $x < y$, iff there exists a path in the net from x to y.
- x and y are in *conflict*, which we denote by $x\#y$, iff there exists two paths $p, t, \ldots, x$ and $p, t', \ldots, y$, starting from the same place $p \in P$ but such that $t \neq t'$.
- x and y are in *concurrency*, which we denote by x co y, iff none of the two previous relations holds, that is to say $\neg(x < y) \wedge \neg(y < x) \wedge \neg(x\#y)$.
- $x \in T$ and $y \in T$ are in *direct conflict*, denoted x conf y, iff they share in their presets the place that originated the conflict ($^\bullet x \cap {}^\bullet y \neq \emptyset$) and $^\bullet x \cup {}^\bullet y$ is a co -set.

We can also find in the literature (*e.g.* [33]) the notion of *weak causality* which is used in the presence of read arcs. However, in our work, we do not consider it.

A set $X \subseteq T$ of transitions are said to be in conflict, noted $\#X$, when some transitions consumed the same token. Formally:

$$\#X = \exists x, y \in X \, , \, x \neq y \wedge {}^\bullet x \cap {}^\bullet y \neq \emptyset$$

The causal past of a transition t is called *local configuration* and denoted by $\lceil t \rceil$. It is constituted by the transitions that causally precede t, *i.e.* $\lceil t \rceil = \{t' \in T \mid t' < t\}$.

Definition 24. ***(Occurrence net)*** An *occurrence net* is an acyclic P/T net $\langle B, E, F \rangle$: DEF

- finite by precedence, *i.e.* $\forall e \in E$, $\lceil e \rceil$ is finite,
- such that each place has at most one input transition, *i.e.* $\forall b \in B$, $|{}^\bullet b| \leq 1$,
- and such that there is no conflict in the causal past of each transition, *i.e.* $\forall e \in E$, $\neg\#\{e \cup \lceil e \rceil\}$. ♦

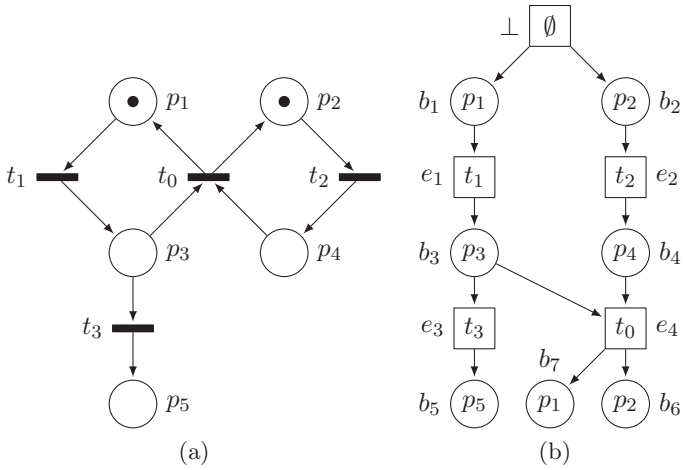


Figure 2.7: A Petri net (2.7a) and its branching processes (2.7b)

We use the classical terminology of *conditions* and *events* to refer to the places B and the transitions E in an occurrence net.

Having the definition of occurrence nets, we define the branching process of a Petri net.

DEF

Definition 25. ***(Branching process of a Petri net)*** A *branching process* of a 1-safe Petri net $\mathcal{N} = \langle P, T, W, M_0 \rangle$ is a labeled occurrence net $\beta = \langle \mathcal{O}, l \rangle$ where $\mathcal{O} = \langle B, E, F \rangle$ is an occurrence net and $l : B \cup E \to P \cup T$ is the labeling function such that:

- $l(B) \subseteq P$ and $l(E) \subseteq T$,
- for all $e \in E$, the restriction $l_{|{}^\bullet e}$ of l to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet l(e)$,
- for all $e \in E$, the restriction $l_{|e^\bullet}$ of l to $e^\bullet$ is a bijection between $e^\bullet$ and $l(e)^\bullet$,
- for all $e_1, e_2 \in E$, if ${}^\bullet e_1 = {}^\bullet e_2$ and $l(e_1) = l(e_2)$ then $e_1 = e_2$.

E should also contain the special event $\perp$, such that: ${}^\bullet\perp = \emptyset$, $l(\perp) = \emptyset$, and $l_{|\perp^\bullet}$ is a bijection between $\perp^\bullet$ and M_0. ♦

EXM

Example 8. Figure 2.7b shows a branching process obtained by unfolding the net presented in Figure 2.7a. The labels are figured inside the nodes. We can see that the branching process in Figure 2.7b unfolds the loop t_1, t_2, t_0 once. This loop could be unfolded infinitely many times, leading to an infinite branching process.

Similarly, we can define the branching process of a network of automata.

Definition 26. ***(Branching process of a network of automata)*** Let $\mathcal{N} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ be network of automata where $\mathcal{A}_i = \langle L_i, l_{0i}, T_i, \Sigma_i \rangle$. A *branching process* of $\mathcal{N}$ is a labeled occurrence net $\beta = \langle \mathcal{O}, l \rangle$ where $\mathcal{O} = \langle B, E, F \rangle$ is an occurrence net and $l : B \cup E \to \left(\bigcup_{i\in[1,n]} L_i\right) \cup Sync$ is the labeling function such that: DEF

- $l(B) \subseteq \bigcup_{i\in[1,n]} L_i$ and $l(E) \subseteq Sync$,
- for all $e \in E$, the restriction $l_{|^\bullet e}$ of l to $^\bullet e$ is a bijection between $^\bullet e$ and $\mathsf{pre}(l(e))$, where $\mathsf{pre}(t) = \{\alpha(t_i) \mid t_i \neq \bullet\}$,
- for all $e \in E$, the restriction $l_{|e^\bullet}$ of l to $e^\bullet$ is a bijection between $e^\bullet$ and $\mathsf{post}(l(e))$, where $\mathsf{post}(t) = \{\beta(t_i) \mid t_i \neq \bullet\}$,
- for all $e_1, e_2 \in E$, if $^\bullet e_1 = {}^\bullet e_2$ and $l(e_1) = l(e_2)$ then $e_1 = e_2$.

E should also contain the special event $\perp$, such that: $^\bullet\perp = \emptyset$, $l(\perp) = \emptyset$, and $l_{|\perp^\bullet}$ is a bijection between $\perp^\bullet$ and $\bigcup_{i\in[1,n]} l_{0i}$. ♦

Branching processes can be partially ordered by a *prefix relation*. For example, if we remove the event e_4 from the branching process in Figure 2.7b, we obtain a prefix of the branching process. There exists the greatest branching process according to this relation which is called the *unfolding* of $\mathcal{N}$.

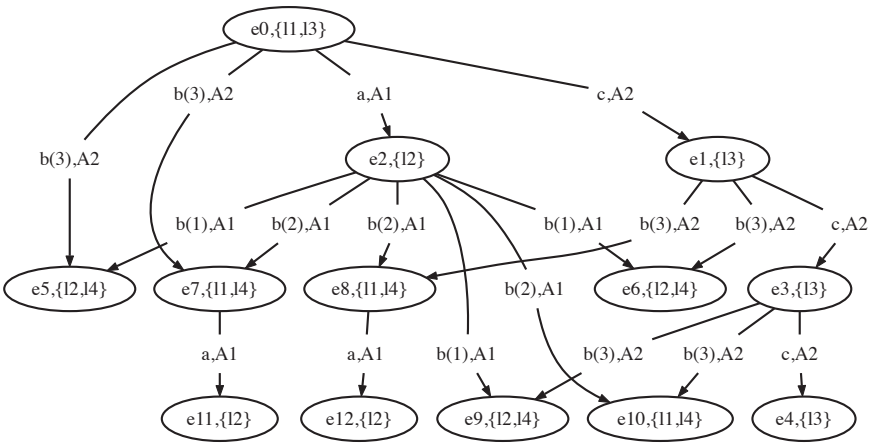
Let $\beta = \langle B, E, F, l \rangle$ be a branching process.

A co-set in β is a set $B' \subseteq B$ of conditions that are in concurrence, that is to say without causal relation or conflict, *i.e.* $\forall b, b' \in B', \neg(b < b')$ and $\neg\# \bigcup_{b\in B'} (^\bullet b \cup \lceil {}^\bullet b \rceil)$.

A *configuration* of β is a set of events $E' \subseteq E$ which is causally closed and conflict-free, that is to say $\forall e' \in E', \forall e \in E,\ e < e' \Rightarrow e \in E'$ and $\neg\# E'$. In particular, the local configuration $\lceil e \rceil$ of an event e is a configuration.

For any co-set B', $l(B')$ defines a subset of the marking of the net. A *cut* is a maximal co-set (inclusion-wise). For any configuration E', we can define the set $\mathsf{Cut}(E') = E'^\bullet \setminus {}^\bullet E'$ which is *e.g.* the marking of the Petri net obtained after executing the events in E'.

An *extension* of β is a pair $\langle t, e \rangle$ such that e is an event not in E, such that $^\bullet e \subseteq B$ is a co-set, the restriction of l to $^\bullet e$ is bijection between $^\bullet e$ and $^\bullet t$ and there is no $e' \in E$ such that $l(e') = t$ and $^\bullet e' = {}^\bullet e$. Adding e to E and labeling e with t gives a new branching process. Starting from the event $\perp$, and successively adding possible extensions form the "unfolding algorithm". In the next chapter, we present the exact procedures for creating branching processes based on both models .



(a)

Figure 2.8: An event structure of the network of automata in Figure 2.1b

Event structure for networks of automata

So far, we presented two definitions of branching processes used for both Petri nets and networks of automata. Nevertheless, in our work, we will sometimes use another, slightly different representation of branching processes of networks of automata which we introduce below.

The event structure which we use for networks of automata consists of events. An event is a vector $e = (e_1, \ldots, e_n)$, where $e_i = (\pi_i(e), \tau_i(e))$ in which $\pi_i(e)$ denotes the predecessor of e considering the automaton $\mathcal{A}_i$, and $(\tau_1(e), \ldots, \tau_n(e)) \in Sync$. In the case where the automaton $\mathcal{A}_i$ is not considered by the transition, we define $\tau_i(e) = \bullet$.

Given two events e and e', e immediately precedes e' in the automaton $\mathcal{A}_i$ (denoted by $e \rightarrow_i e'$) if $\pi_i(e') = e$. A set E of events is in conflict iff $\exists e, e' \in E, i \in [1, n]$ such that $\pi_i(e) = \pi_i(e')$. From the causality point of view, none of the events can have conflicts in its causal history as it would mean that there are two exclusive events. Such events are the result of a local choice of a single automaton.

EXM

Example 9. Figure 2.8 presents an example of an event structure of the network in Figure 2.1b.

Graphically, an event e is represented by a node, with an incoming arc from each node $\pi_i(e)$ labeled by $\langle \lambda(\tau_i(e)), \mathcal{A}_i \rangle$. Each node is represented by an ellipse, labeled with the name of an event e and the local states reached by the corresponding transition.

Symbolic occurrence nets

So far, the structures that we presented above do not include time constraints. More details of construction of such structures are given later in Chapter 3. Below, we only briefly mention some of the main concepts connected with the structures.

As we presented before, both time Petri nets and networks of timed automata in addition to the discrete transitions also have the delay transitions. Let us consider an example of a system which produces some events, each of which has a simple timestamp assigned to it. Each timestamp represents a clock valuation. With such knowledge, it is easy to imagine that, even by having only one transition in a timed model, it may be associated with (possibly infinitely) many events in reality. That is why equipping events with a timestamp is usually not practical. And also for this reason so called *symbolic occurrence nets* were introduced.

In symbolic occurrence nets instead of events, we operate on symbolic events that are events which in fact represent a number of simple events. A usual way to define such a structure is to use some constraints on time rather than the explicit timestamps. As we see in Chapter 3 the form of these constraints above all depends on the model we consider.

The subject of *symbolic occurrence nets* was already studied in several articles, both in the context of time Petri nets ([34]) and networks of timed automata ([29, 26]).

2.6.2 Observations

The process of observation is an important source of information about events that occur or could occur in the system. Monitoring of distributed systems is often hampered by the lack of global references, such as time, or information about the order of events. As we will see in the next chapter, in many cases despite the absence of such data, it is possible to deduce interesting information about the possible behavior of events in the system.

We consider that the real distributed system under supervision has been instrumented in such a way that it will produce events (like prints used for debugging) during its execution. These events have a name picked up in some finite alphabet Σ. In order to relate the observation and the model, we also consider that transitions of the model are labelled by a similar function $\lambda : T \to \Sigma \cup \{\epsilon\}$. The ϵ symbol not belonging to Σ is used to indicate that the occurrence of the transition cannot be linked to an observable event. The labeling does not need to be injective, and in general, the same observation can be explained by several trajectories of the model.

Below, we present a brief description of the observations and the way they may affect process of monitoring which is of our interest. We start with a presentation of three basic types of observations. The type of observation

depends on the sort and amount of information which are provided in the monitoring process. We will distinguish the following types of observations: unstructured observations, structured observations, and observations with timestamps.

Unstructured observations

DEF **Definition 27.** ***(Unstructured observation)*** An *unstructured observation* is a finite set of events. Each event has a name given by the labeling function $\lambda : O \to \Sigma$.

In the case of unstructured observations, the only available information about events is their name which may usually be attributed to a specific transition in the underlying model of the system.

It can easily be noted that unstructured observations are the least restrictive type of observations. During monitoring process, this type of observations may cause numerous questions about the possible behaviors of the considered system. This is mainly due to the fact that the set of events forming this type of observation is completely unordered. This lack of information makes the number of possible scenarios corresponding to the observation dramatically large.

In other words, we are talking about unstructured observations when there is no information about causal relations between the events that are part of the observation. This situation is not uncommon in distributed systems. Of course, there are many situations in which determination of the order between events is possible. Then, however, we have to deal with the second type of observation, *i.e.* structured observations.

Structured observations

Even if it is sometimes difficult for distributed systems, there are a lot of situations when the order of the events can be determined. A simple example would be *e.g.* well-known vector clocks ([47, 69]). Sometimes, in order to determine the order of events in a distributed system, the real-time can also be used. Unfortunately, such an information is sometimes not accurate enough.

As we can expect, structured observations provide more information than unstructured observations. In this case, the information about events which are flowing from the system to the supervising system is enriched with additional data about causal relations between events. As we can expect, such an information has great impact on searching and analyzing the trajectory of the system.

DEF **Definition 28.** ***(Structured observation)*** An *observation* is a finite set of events O equipped with a causal order $\preceq$ and a symmetric relation *co*. If

two events are not related, their relation is said “unknown". An event also has a name, given by the labeling function $\lambda : O \to \Sigma$.

In the above definition, we can note that we distinguish three types of relations that can occur between events in the observation (see also Section 1.3). Thus, we can consider the following three cases: two events can be causally related, they can be concurrent, or their relation is not known. As usual, the causal relation must be an order. The two others are just symmetric.

Later, in Chapter 3, we will see that this type of observation can be used to guide the construction of a finite unfolding containing the configurations that are *compatible* with the observations. Intuitively, we consider the maximal configurations and ask if they do not contain events and relations that contradict the observation.

Observations with real-time timestamps

As mentioned above, apart from knowing the name of an event, sometimes we can also obtain information about the time in which the event occurred. A precise knowledge about execution time of an event is a very valuable information from the viewpoint of the monitoring system. Such an information can frequently help to specify causal relations between events. The problem starts when we realize that the information about time is not perfect, which is almost always the case in real-time distributed systems. In order to reduce errors in measurement of time, there are obviously many additional solutions such as clock synchronization protocols. But even they do not always give satisfactory results. Nevertheless, it may turn out that even this inaccurate information about the time can be useful. To solve such a problem, a given system can be split, for instance into smaller groups within which the measurement of time is sufficiently accurate to define causal dependencies between events.

Example 10. To give a simple idea of how the three types of observation may look like, we present an example observation of four events a, b, c, d. For the unstructured observation, we could have a set $\{a, b, c, d\}$, for the structured observation $\{(a \prec b \prec c), (b \prec d)\}$, and finally for observations with real-time timestamps $\{(a, 1), (b, 2), (c, 4), (d, 3)\}$ in which each pair consists of the event name and its timestamp. EXM

Note that having only the observation with timestamps, it is not always possible to deduce the causal dependency between the events (unless we know that, for example, the events belong to the same process). In our example, c happens after d, but this fact does not mean that $d \prec c$.

Obviously the types we mentioned above are not the only possibly ones. We can imagine, for instance, a simple mix of them where a part of the observation is structured and another part is with some real-time information.

Our classification certainly does not cover all possible types of observations, such as probabilistic observations. However, in our work we mainly focus on the unstructured observations as they are the most general case in terms of monitoring in some sense. The rest of the cases, *i.e.* structured observations or observation with timestamps, usually implies a set of explanations which is in fact a subset of explanations obtained for unstructured observations.

2.6.3 Explanations

As we already know, the main task of the supervisory system is to track selected events occurring in the system and then to create scenarios based on them (referred to as *explanations*) and compatible with the observations.

As we described in the previous section, observations that come to supervisory system are usually incomplete, in the sense that they do not let unequivocal reconstruction of the behavior of the monitored system. Among factors which hinder creation of the explanations by the monitoring system, we may find, for example:

- lack of traceability of all events, *e.g.* as a result of the so-called unobservable events. Let us recall that these are events that are visible in the model in the form of transitions which are usually marked with the symbol ϵ. However, we can not observe them during activity of the system. The presence of such events is one of the reasons why explanation for a finite number of observations can be infinite, or the number of explanations can be infinite. As we will see in Chapter 4, this happens because the model may contain unobservable loops, that are loops containing only ϵ-transitions. In the following part, we also see that the number of unobservable transitions in the system often significantly increases the number of possible explanations.
- lack of causal order of events in observations (see Section 2.6.2).
- lack of unambiguous information about events. This makes assignment of events to specific transitions in the model difficult or even impossible. For example, such a situation can take place when several transitions in the model have identical labels.

We already know that appropriate structures are necessary for storing and processing explanations. In our case, the natural choice is the already mentioned symbolic branching processes (Section 2.6.1), and the special event structures dedicated for networks of automata. As we described earlier, these structures enable us to maintain information about causal dependencies between events. If a branching process contains all possible explanations for a given observation, we call it *complete*. Thus, as we will see in Chapters 3 and 4, a single branching processes can store many different explanations which

are *compatible* with a given observation. In short, in order to construct such a branching process, the supervisory system *unfolds* the concerned model with respect to the observations. The resulting structure is called *constrained unfolding*. It is worth noting that explanation with the same prefixes can share with each other the common prefix, thereby saving memory.

When we consider models with time constraints, it appears that one of their key features is the possibility to represent in symbolic way information about time of execution of events.

As we will see in section on constrained unfoldings, there is also a possibility to use parameters in time constraints. This often helps to obtain the information about the possible execution time of events in the system.

Chapter 3

Constrained unfoldings

3.1 Introduction

In this chapter we discuss a dynamic verification method called *model-based supervision*. It is established that diagnosing dynamical systems, represented as discrete-event systems, amounts to finding what happened to the system from existing observations (an event log) derived from sensors. In this context, the diagnostic task consists in determining the trajectories compatible with the observations. The standard situation is that the observed events correspond to the firing of some transitions of the model, while the other transitions are just internal (this situation is called "partial observation" in supervisory control theory [28]).

Among the different analysis techniques, we chose to develop the work on unfoldings [44]. The great interest of unfoldings in that task is their ability to infer the possible causal dependencies which in general are not part of the observations. Unfoldings were introduced in the early 1980s as a mathematical model of causality and became popular in the domain of computer aided verification. The main reason was to speed up the standard model-checking technique based on the computation of the interleavings of actions, leading to a very large state space in case of highly concurrent systems. The seminal papers are [70] and [42]. They dealt with basic bounded Petri nets. Since then, the technique has attracted more attention, and the notion of unfolding has been extended to more expressive classes of Petri nets (Petri nets with read and inhibitor arcs [12, 33], unbounded nets [3], high-level nets [60], and time Petri nets [35]).

Supervision, based on unfoldings in our case, is implemented by the on-the-fly construction of the unfolding, guided by the observations. With this dynamic approach, since we only consider finite sequences of observations, decidability questions become much easier. The only requirement is to be able to decide whether a transition can be fired or not. Note that the lack of existence of a finite prefix in the stopwatch ([21]) or parametric ([86]) cases is

not necessarily prohibitive as several analysis techniques, such as supervision, can do without it. Practical experience also demonstrates that, even for very expressive models such as Linear Hybrid Automata [57], the undecidability of the interesting problems still allows to analyze them in many cases.

We split the chapter into several parts according to the type of model we deal with, as it is done in the previous chapter.

We start our presentation in the chapter with networks of timed automata. We describe a unified way to process the model of networks of timed automata (introduced in [52]) without computation of any intermediate structures such as a single automaton representing all the components. Next we show how to apply the unfolding method to solve a supervision problem.

In the following sections 3.4 and 3.5, we present the similar subject of supervision step by step but in the context of time parametric Petri nets.

Petri nets for supervisory control and diagnosis have been proposed in numerous papers (see for instance [87] and [51]). In most cases, the construction of diagnosers is based on the state graph (*i.e.* the interleaving view). The use of unfoldings is more recent. [46] uses safe ordinary nets and focuses on the distributed diagnosis, [34] proposes to use unfolding of safe time Petri nets. The parametric case has not been considered yet.

Advancing this line, we present a method to unfold safe time parametric Petri nets (described in [85] and [53]) in this chapter. The two novelties are: a new unfolding algorithm, as an optimistic alternative to [35], and its natural extension to deal with parameters. We think that adding parameters in specifications is a real need. It is often difficult to fix them a priori: indeed, we expect from the analysis some useful information about their possible values. This feature of genericity clearly adds some "robustness" to the modeling phase. It is particularly relevant for the supervision activity we consider, in which an arbitrary choice of parameters often avoids to find explanations compatible with the observations. This leads to the rejection of the model. Moreover, no additional knowledge about the way to correct it is provided.

For both models, we also give some case studies such as the classical alternating bit protocol in which timeout duration and messages delays can be parameterized. It shows the ability of the method to infer from a simple sequence of observed events, explanations in which we explicitly see the reordering of messages and we automatically infer a required condition linking the timeout constant and the communication delays.

Finally, in Section 3.6, we briefly present some final remarks.

3.2 Supervision of untimed networks of automata

In the following sections, we present the basics of the unfolding techniques for networks of timed automata step by step. At the same time, we introduce the concept of constrained unfoldings which is the target structure used for supervision. We follow the description of the models presented in Chapter 2. This way, the reader can follow the notion of constrained unfoldings for models with different features.

3.2.1 Finite state automaton

We start with the simplest version of automaton that is a finite state machine (see Definition 3).

It is known that all executions of such an automaton can be represented as labeled paths in a tree graph with the initial state as the root. Such labeled paths can be formally defined with the use of notion of unfolding.

Let us shortly recall that we denote a transition of a finite automaton by $t = (\alpha(t), \lambda(t), \beta(t))$, *i.e.* $\alpha(t)$ is the initial state of the transition, $\lambda(t)$ is a label assigned to the transition t, $\beta(t)$ is the final state of the transition.

An unfolding of an automaton $\mathcal{A}$, denoted by $\mathcal{U}(\mathcal{A})$ is given by a set of events (see Section 2.6.1). Each event represents an occurrence of a transition. It is defined by a pair $e = (\pi(e), \tau(e))$, where $\pi(e)$ is the event which precedes e in the unfolding, and $\tau(e)$ is the transition assigned to e. There is also a fictitious initial event $\perp$ for which $\beta(\tau(\perp)) = q_0$.

$\mathcal{U}(A)$ can inductively be defined as follows.

DEF

Definition 29. ***(Unfolding of finite automaton)*** Given a finite state automaton $\mathcal{A} = \langle Q, q_0, T, \Sigma \rangle$, the unfolding of $\mathcal{A}$, denoted by $\mathcal{U}(\mathcal{A})$, is the smallest set such that

- $\perp \in \mathcal{U}(\mathcal{A})$, and
- $\{\exists \pi(e) \in \mathcal{U}(\mathcal{A}) \wedge \exists t \in T \wedge \alpha(t) = \beta(\tau(e))\} \implies (e, t) \in \mathcal{U}(\mathcal{A})$ ♦

Having two events e and e' of $\mathcal{U}(\mathcal{A})$, e immediately precedes e' (denoted by $e \to e'$) if $\pi(e') = e$. The causality between two events is defined as a reflexive and transitive closure of the relation $\to$ (denoted by $\to^*$). For an event e, its set of causal predecessors is denoted by $\downarrow e = \{f \mid f \to^* e\}$. This notation is extended to sets: $\downarrow E = \bigcup_{e \in E} \downarrow e$.

In general, unfoldings are infinite sets, *e.g.* unfoldings of automata with loops.

EXM

Example 11. In Figure 3.1, there is a finite subset of the unfolding of the automaton in Figure 2.1a. The subset is closed by precedence relation and is called a prefix of the unfolding. Graphically, an event $(\pi(e), \tau(e))$ is represented by a node with an incoming arc labeled by $\lambda(\tau(e))$ which

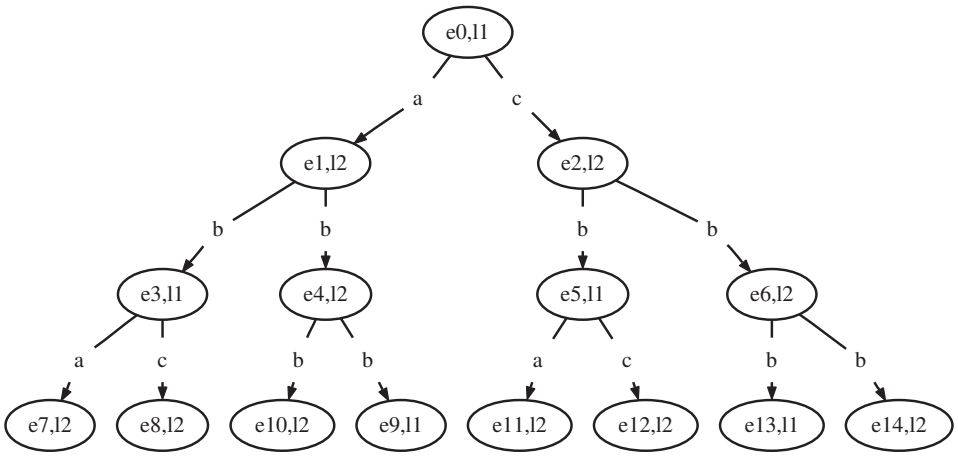


Figure 3.1: A prefix of the unfolding of the automaton in Figure 2.1a.

starts from the node $\pi(e)$. The events are drawn as ellipses which contain the name of the event and the reached state.

The prefixes of unfoldings of finite state automata are trees with bounded degree.

3.2.2 Network of finite state automata

As in the case of a single automaton, an unfolding of a network of automata $\mathcal{N}$ (see Definition 4), denoted by $\mathcal{U}(\mathcal{N})$, is given as a set of events. An event is a vector $e = (e_1, \ldots, e_n)$, where $e_i = (\pi_i(e), \tau_i(e))$ in which $\pi_i(e)$ denotes the predecessor of e considering the automaton $\mathcal{A}_i$, and $(\tau_1(e), \ldots, \tau_n(e)) \in Sync$. In the case where the automaton $\mathcal{A}_i$ is not considered by the transition, we define $\pi_i(e) = \epsilon$.

Given two events e and e', e immediately precedes e' in the automaton $\mathcal{A}_i$ (denoted by $e \rightarrow_i e'$) if $\pi_i(e') = e$. A set of events is in conflict iff $\exists e, e' \in E, i \in [1, n]$ such that $\pi_i(e) = \pi_i(e')$. From the causality point of view, none of the events can have conflicts in its causal history as it would mean that there are two exclusive events. Such events are the result of a local choice of a single automaton.

DEF **Definition 30.** ***(Unfolding of a network)*** Given a network $\mathcal{N}$, $\mathcal{U}(\mathcal{N})$ is the smallest set satisfying:

- $\perp \in \mathcal{U}(\mathcal{N})$, and
- $$\begin{cases} (\tau_1(e), ..., \tau_n(e)) \in Sync \\ \forall i \in [1,n] \begin{cases} \tau_i(e) = \epsilon \Rightarrow \pi_i(e) = \epsilon \\ \tau_i(e) \neq \epsilon \Rightarrow \begin{cases} \pi_i(e) \in \mathcal{U}(\mathcal{N}) \\ \alpha(\tau_i(e)) = \beta(\tau_i(\pi_i(e))) \end{cases} \end{cases} \\ \downarrow e \text{ is conflict free} \end{cases}$$

$\Longrightarrow\ e \in \mathcal{U}(\mathcal{N})$ ♦

Example 12. Figure 3.2b presents an example of a prefix of unfolding of the network in Figure 3.2a. The set of synchronizations is $\{(a, \epsilon),\ (b(1), b(3)),\ (c, \epsilon),\ (b(2), b(3))\}$. The initial event $\perp$ is represented as e_0 in this and all the following figures. Inside each ellipse which denotes an event apart from its name, there is a set of locations which is reached after firing the corresponding transitions of an event. For example, for the event e_6, there are two local transitions $b(1)$ and $b(3)$ which take part in the action b. After this action the network reaches the locations l_1 and l_4. EXM

Graphically, an event e is represented by an arc going from the node $\pi_i(e)$ to the node e, and labeled by $(\lambda(\tau_i(e)), \mathcal{A}_i)$. Each node is represented by an ellipse, labeled with the name of an event e and the local states reached by the corresponding transition.

In general the prefixes of unfoldings of networks are acyclic graphs with an unbounded degree. The inductive definition can directly be used to construct an algorithm in which the events are placed one by one in the unfolding if they are not already there.

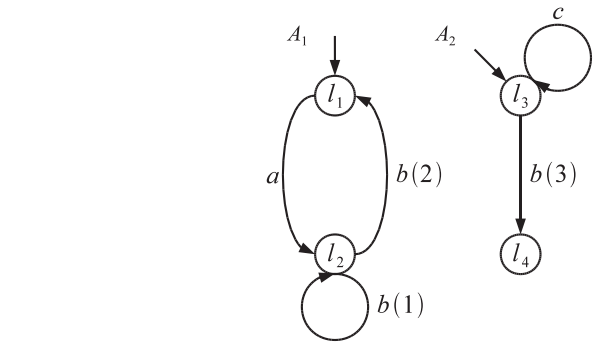
Constrained unfolding

Having the notion of unfolding, the question of supervision can be addressed. We consider a sequence of observations $\sigma \in \Sigma^*$. The problem is to construct all possible executions of a network which are correct with respect to the observations. In other words, we search for explanations of what is observed. The sequence of observations is finite, and so is the unfolding.

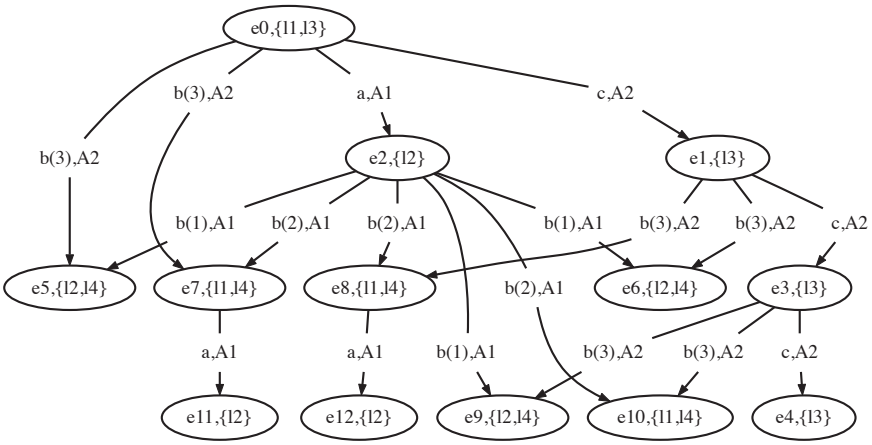
The idea is that the order given by the sequence of observations σ does not necessarily correspond to the real order of the corresponding events. The observation is just the result of the monitoring process in the distributed system that is observed. It is the role of the supervisor using the model to propose the possible causal relationship between observations. It is clear that the actions of the same type are totally ordered as they correspond to a local action of an automaton or a synchronization. A good method to guide a construction of an unfolding is to use the Parikh function of the sequence σ (for more details see e.g. [44]).

Definition 31. ***(Parikh function)*** Let $\sigma \in \Sigma^*$ be a sequence of observation. The Parikh function $\varpi : \Sigma^* \to \mathbb{N}^{|\Sigma|}$ counts the number of occurrences of each symbol of the sequence. ♦ DEF

We extend the information of an event e by the Parikh vector $\varsigma(e)$ of the sequence recognized by the set of causal predecessors of e. By comparison of the Parikh vector of an event with the Parikh vector of an observation, we ensure that the latter does not exceed the former. For an action $a \in \Sigma$, we denote by χ_a the Parikh vector which has all its components set to 0



(a)



(b)

Figure 3.2: A network of finite state automata (a) and a prefix of the unfolding of the network (b).

except the one which corresponds to the action a and equals 1. The events are denoted by $e = \left((\pi_i, \tau_i)_{i\in[1,n]}, \varsigma(e)\right)$.

Example 13. Let us take a prefix in Figure 3.1 and event e_5. We can observe that the event e_5 is preceded by two transitions labeled by b and c. Thus, the observation produced by the predecessors of e_5 is $\{b, c\}$. This implies, in turn, $\varsigma(e_5) = \varpi(\{b, c\}) = [0, 1, 1]$, where the subsequent positions in the vector denote the number of occurrences of events respectively labeled by a, b, and c. EXM

The unfolding guided by the observation, denoted as $\mathcal{E}(\mathcal{N}, \sigma)$, may therefore be defined as follows.

Definition 32. ***(Constrained unfolding of network of automata)*** Given a network $\mathcal{N}$ and a sequence of observations σ, a *constrained unfolding* of $\mathcal{N}$, denoted by $\mathcal{E}(\mathcal{N}, \sigma)$, is the smallest set satisfying the following conditions: DEF

- $\perp \in \mathcal{E}(\mathcal{N}, \sigma)$ with $\varsigma(\perp) = \mathbf{0}$, and
- $$\left\{\begin{array}{l} t = (\tau_1(e), ..., \tau_n(e)) \in Sync \\ \forall i \in [1, n], \left\{\begin{array}{l} \tau_i(e) = \epsilon \Rightarrow \pi_i(e) = \epsilon \\ \tau_i(e) \neq \epsilon \Rightarrow \left\{\begin{array}{l} \pi_i(e) \in \mathcal{E}(\mathcal{N}, \sigma) \\ \alpha(\tau_i(e)) = \beta(\tau_i(\pi_i(e))) \end{array}\right. \end{array}\right. \\ \downarrow e \text{ is conflict free} \\ \varsigma(e) = \sum_{f\in\downarrow e} \chi_{\lambda(\tau(f))} \leq \varpi(\sigma) \end{array}\right.$$
  $$\Longrightarrow e \in \mathcal{E}(\mathcal{N}, \sigma)$$ ♦

Note that the definition of constrained unfolding of network of automata is quite similar to the definition of unfolding. The key condition in this case, *i.e.* $\varsigma(e) \leq \varpi(\sigma)$, verifies whether the predecessors of e do not exceed the observation σ.

Example 14. Figure 3.3 shows the constrained unfolding obtained for the sequence of observations $\sigma = acbc$ for network in Figure 3.2a. The vector $\varsigma(e)$ is shown next to each event. For the sake of simplicity, it is represented as a set of equalities of the form $x = y$, where x stands for the number of transitions labeled by x and y is the corresponding value. In the figure, we distinguished three events (dashed border) e_4, e_{11}, e_{12} for which $\varsigma(e) \not\leq \varpi(\sigma)$, where $e \in \{e_4, e_{11}, e_{12}\}$. For example, for e_{12}, the number of transitions labeled by a is greater than 1, which is the case in σ. EXM

The result represents the two following explanations: the actions a and two c are carried out independently and then the action b is fired. In the case of the action b there are two possibilities as there are two actions labeled by b in the first automaton. These explanations are obtained by:

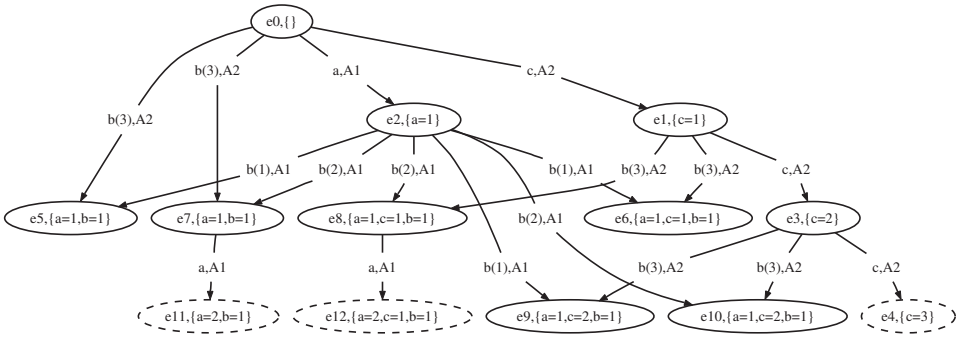


Figure 3.3: A constrained unfolding.

- extracting executions of the unfolding. An execution is defined by a subset of E of the unfolding $\mathcal{E}(\mathcal{N}, \sigma)$ which is closed by causality ($\downarrow E = E$) and without conflicts; and
- requiring that the executions explain the whole sequence of observations, *i.e.* $\sum_{e \in E} \chi_{\lambda(\tau(e))} = \varpi(\sigma)$.

As we could see in the example above, not only we infer the trajectories of automata, but also the possible causal links between observations. This is what gives a real meaning of the method in the context of supervision, thanks to which we get the extra information to the sequence of observations.

3.2.3 Problem with partial observation

In many cases, we cannot or we do not want to observe all the transitions of the model. For this reason, we introduce a construction of an unfolding using only a sequence of partial observations. To construct such an unfolding, we assume that we observe only some transitions of the network. They can be any transitions of the model. We can observe some local transitions or some global transitions which involve several automata. By a global transition we mean a synchronization which we introduce to emphasize the fact that we treat it as a single non-local transition, and not as a collection of local transitions (see Definition 4).

For the construction of an unfolding based on some partial observation σ, we slightly modify the technique used in Definition 32. To mark unobservable transitions, we use a boolean function $\nu(x)$, where x is an event or a transition. An event e is observable (*i.e.* $\nu(e)$ is true) if at least one of its underlying transitions is observable. Formally, we can write it as follows:

$$\nu(e) \iff (\exists \tau_i \in \tau(e), \nu(\tau_i)) \vee \nu(\tau(e))$$

Note that, in this definition, we distinguished two situations: one when a local transition is observable or not ($(\exists \tau_i \in \tau(e), \nu(\tau_i))$ is true or false),

and the other one when the global transition is observable or not ($\nu(\tau(e))$ is true or false). We have made a distinction in order to use a slightly modified version of the method with the Parikh vector. Namely, in the vector we put an extra position which is a number of events that cannot be observed during the considered execution. Thus, during the construction of the unfolding, every time an event which is not observable is produced, the value of the mentioned position is increased. On the contrary, whenever there is an observable event e, we have to increase the value which corresponds to the transition $\tau(e)$. However, in the latter case, we may not observe the whole global transition if it consists of several local transitions and if some of them are not observable. Then, we increase only the values of the vector which are assigned to the local observable transitions. This way, when we observe a whole global transition (*i.e.* all its underlying local transition), only one position of the vector is modified. Such approach lets us easily compare a sequence of observations and a set of events produced by an execution in the unfolding.

It is important to mention that one of the main problems, when we want to infer some information about a system which is only partially observable, is that there may be infinite loops which are unobservable. For this reason, in our solution, we bound the number of unobservable events for each possible execution E of the system, *i.e.* $|\{e \in E \mid \neg\nu(e)\}| \leq M$. We deal with the topic of unobservable loops more precisely in Chapter 4.

Example 15. In Figure 3.4, we present an example of the constrained unfolding of the system in Figure 3.2. In the construction of the prefix, an extra parameter is used, which limits the number of unobservable events in the set of predecessors of any event (in the example, it is set to 2). The description of events is the same as in Figure 3.3. ? denotes the number of unobservable events in the set of predecessors of the considered event. The dotted ellipses and arrows denote events and transitions which are unobservable. The events e_{10}, e_{11}, e_{12} represent possible explanations for the given sequence of observations. EXM

In the example, we observe two transitions: a and $b(3)$. It is worth noting that we do not observe the whole action b but just one of the local transitions which take part in it. Thus we do not care and we cannot distinguish which of the two possible synchronizations takes place. Let us take one of the successful events (*i.e.* a maximal event of a valid explanation) from Figure 3.4 ,*e.g.* e_{12}. As an input we have a sequence of observations *aba*. We can notice that, among predecessors of e_9, there are two observable events $\{e_2, e_9\}$ (we do not count the initial event $e_0 \equiv \perp$) and two unobservable events $\{e_1, e_3\}$. When we take all the transitions of the events $\{e_1, e_2, e_3, e_9, e_{12}\}$, we get the vector in which we have two unobservable events (in the figure $? = 2$), two transitions a, and one $b(3)$.

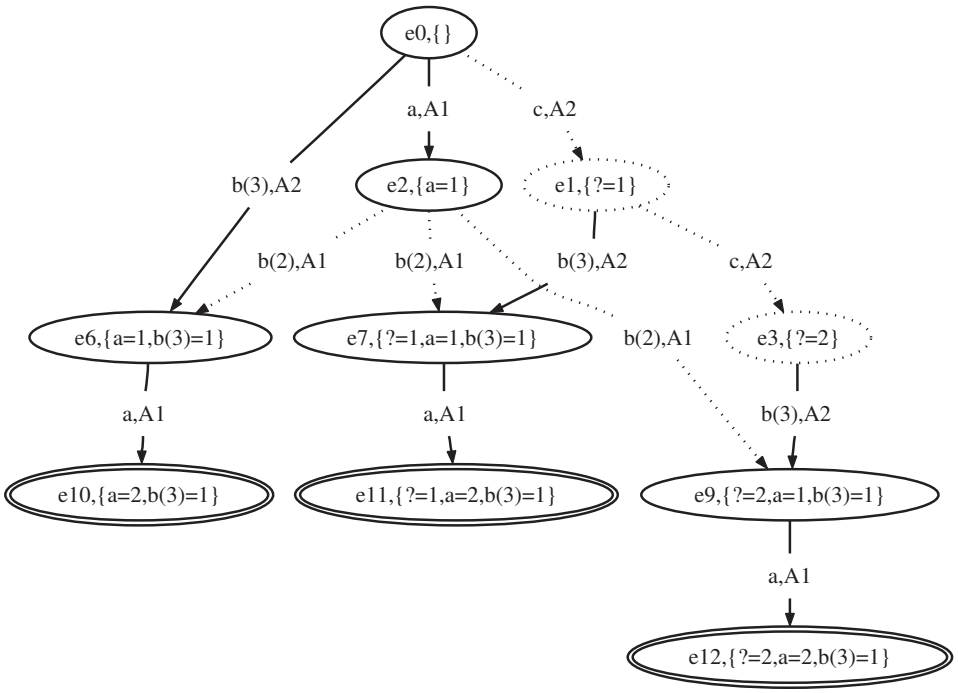


Figure 3.4: A prefix of unfolding of the network in Figure 3.2 guided by the partial observation *aba*.

3.3 Supervision using networks of timed automata

3.3.1 Constrained unfoldings of timed automata

The question of supervision of timed systems can be placed in the same context as in the case of systems without time constraints. Let us imagine a sequence of observations made only of a series of symbols of the alphabet Σ. The problem is to find executions that can explain this sequence. What is especially interesting is to infer the causal relationships induced by the model.

The first idea is to proceed as in the untimed case and to define the notion of unfolding of a network of timed automata. The sequence of observations can also naturally be seen as a timed automaton without time constraints. The notion of timed unfoldings has recently been introduced in [29]. However, its calculation is complex. In the obtained structure, each of the events is assigned a symbolic expression which gives the possible dates of firing transitions. The concept of conflict also has to be weakened in the notion of asymmetrical conflict in order to keep concurrency.

We propose a simpler approach to answer the question of supervision. By definition, the introduction of time constraints limits the possible executions of the model. Thus, we can consider the explanations produced for the un-

derlying untimed model (*i.e.* the unfolding guided by the observations) and then take into account the time constraints. Furthermore, we will see that this phase of post-selection allows inferring the possible dates of observation. This is a potentially rich information for supervision activities.

Thus let us consider a network of timed automata, its untimed underlying network $\mathcal{N}$ and a sequence of observations σ. Let us also take the set of possible untimed explanations, *i.e.* all the sets of events $E \subseteq \mathcal{E}(\mathcal{N}, \sigma)$ such that $\downarrow E = E$ is without conflict and $\sum_{e \in E} \chi_{\lambda(\tau(e))} = \varpi(\sigma)$.

We consider an explanation E. We denote by E_i the set of events concerned by the automaton $\mathcal{A}_i$ (*i.e.* the events e such that $\tau_i(e) \neq \epsilon$). We know that the closure $\rightarrow_i^*$ is a total order on E_i since the processes are sequential. We will denote by $\uparrow_i E$ the maximal event for this relation. For each event, we will denote by $\delta(e)$ a date of the event and by $dor_i(e)$ a date of reseting clocks (X_i) of the automaton $\mathcal{A}_i$ after the event occurred. $dor_i(e)$ is defined as follows:

$$\forall x \in X_i, \forall e \in E, dor_i(e)(x) \stackrel{def}{=} \begin{cases} \delta(e) & \text{if } x \in \rho(\tau_i(e)) \\ dor_i(\pi_i(e))(x) & \text{otherwise} \end{cases}$$

Definition 33. ***(Time validity)*** An explanation E which does not take time into consideration is valid according to the time constraints of the network iff: DEF

$$\forall i \in [1,n] \begin{cases} \forall e \neq \perp \in E_i \begin{cases} \delta(\pi_i(e)) \leq \delta(e) \\ I_i(\alpha(\tau_i(e)))(\delta(e) - dor_i(\pi_i(e))) \\ \gamma(\tau_i(e))(\delta(e) - dor_i(\pi_i(e))) \end{cases} \\ I_i(\beta(\tau_i(\uparrow_i E)))(max_{f \in E}\delta(f) - dor_i(\uparrow_i E)) \end{cases}$$

♦

The definition of time validity incorporates the definition of sequential semantics. We explain the formula line by line:

- the time cannot go back between the causal predecessor of an event and itself;
- the invariants of the initial locations of the transition are satisfied at the moment of firing the transition;
- the guards of the local transitions which form the transition of the considered event are also satisfied at the time of firing; and
- the invariants of the final locations of the transition are satisfied at the end of the explanation.

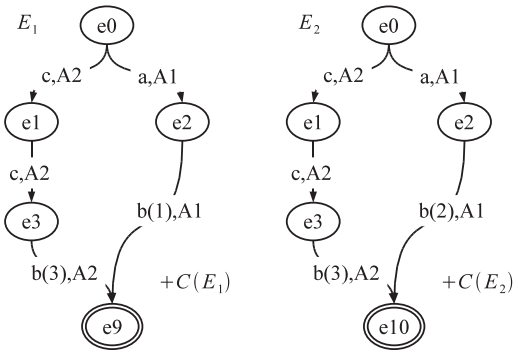


Figure 3.5: Two possible explanations E_1 and E_2 with time constraints $C(E_1), C(E_2)$ on dates of occurrences of events.

These conditions can therefore reject the explanations which are invalid if time constraints are taken into consideration. Beyond that, we can pass to the symbolic representation constraints having in mind that, for each event e, $\delta(e)$ is a real variable assigning the possible dates of firing the corresponding transition of the event e.

EXM **Example 16.** In the example presented in Figure 2.3, we have a network of automata which produces four actions a, b, and two actions c. The possible explanations of these actions are shown in Figure 3.5. What we want to know is the possible dates of these actions (denoted $\delta(a), \delta(b)$ and $\delta(c)$). As we can see, the action c can only be produced at time 1. After this action, we can reset the clock and repeat it at time 2. After these two actions, the automaton $\mathcal{A}_1$ fires action b but , for that purpose, it has to be synchronized with the second automaton $\mathcal{A}_2$. Thus the automaton $\mathcal{A}_1$ has to go to the location l_2 before the action c and then it can choose between two transitions labeled by b. If the automaton executes the action b that produces the state in which automaton $\mathcal{A}_1$ is in location l_1, we know that $\delta(a) = \delta(b)$ because of the guard $x = 0$. We actually have $\delta(b) \leq 3$, otherwise we would have another action c. Moreover, since we know that the global time progresses, we can infer that $\delta(b) \in [2, 3]$. In the other case, where the transition b was the effect of the synchronization of $b(1)$ with $b(3)$, we have $\delta(b) \leq 3$ like previously but also $\delta(b) \leq \delta(a) + 1$. Otherwise, another action c would be produced. In fact we can deduce that $\max(\delta(a), 2) \leq \delta(b) \leq \min(\delta(a) + 1, 3)$. This is the type of information that we will try to automatically infer during the supervision.

For our example in Figure 3.5, the two possible explanations are described with the following constraints:

- $C(E_1) \equiv \begin{cases} \delta(\perp) \leq \delta(e_1) \\ \delta(e_1) \leq \delta(e_3) \wedge \delta(e_3) \leq \delta(e_9) \\ \delta(\perp) \leq \delta(e_2) \wedge \delta(e_2) \leq \delta(e_9) \\ \delta(e_9) - \delta(e_3) \leq 1 \wedge \delta(e_9) - \delta(e_2) \leq 1 \\ \delta(e_3) - \delta(e_1) \leq 1 \wedge \delta(e_3) - \delta(e_1) = 1 \\ \delta(e_1) - \delta(\perp) \leq 1 \wedge \delta(e_1) - \delta(\perp) = 1 \\ \max(\delta(\perp), \delta(e_1), \delta(e_2), \delta(e_3), \delta(e_9)) - \delta(e_2) \leq 1 \end{cases}$

- $C(E_2) \equiv \begin{cases} \delta(\perp) \leq \delta(e_1) \\ \delta(e_1) \leq \delta(e_3) \wedge \delta(e_3) \leq \delta(e_{10}) \\ \delta(\perp) \leq \delta(e_2) \wedge \delta(e_2) \leq \delta(e_{10}) \\ \delta(e_{10}) - \delta(e_2) \leq 1 \wedge \delta(e_{10}) - \delta(e_2) = 0 \\ \delta(e_{10}) - \delta(e_3) \leq 1 \wedge \delta(e_3) - \delta(e_1) \leq 1 \\ \delta(e_3) - \delta(e_1) = 1 \wedge \delta(e_1) - \delta(\perp) \leq 1 \\ \delta(e_1) - \delta(\perp) = 1 \end{cases}$

After reduction and with assumption that $\delta(\perp) = 0$, we obtain:

$$C(E_1) \equiv (e_1 = 1) \wedge (e_3 = 2) \wedge (2 \leq e_9 \leq 3) \wedge (0 \leq e_9 - e_2 \leq 1),$$

and

$$C(E_2) \equiv (e_1 = 1) \wedge (e_3 = 2) \wedge (2 \leq e_{10} \leq 3) \wedge (e_{10} - e_2 = 0)$$

which very well confirms the previous informal analysis of the executions of the timed model.

3.3.2 Case study

In the following case study, we present a network of timed automata which models the alternating bit protocol. The whole model consists of four automata. They are illustrated and described in Figure 3.6. A_S is a sender, A_R a receiver, and a parametrized automaton $A(x, y)$ which represents two analogical automata $A_{(S \to R)}, A_{(R \to S)}$ used to model communication channels, each for one direction. $A_{(S \to R)}$ denotes the communication channel from the sender to the receiver, whereas $A_{R \to S}$ is used for the opposite direction. Dotted lines denote unobservable transitions. We can only observe transitions which are drawn with a solid line. Therefore, we assume that the user of the protocol can only observe four transitions of the model, *i.e.* $!m_0(0)$ which is the first emission of a message with bit 0, $!m_1(2)$ which is the first emission of a message with bit 1, $?m_0(0)$ and $?m_1(1)$ which are the first receptions of bit 0 and 1 respectively. It is worth noting that the user cannot observe the retransmissions ($!m_0(2)$ and $!m_1(0)$). This is a task of the protocol to retransmit and manage all of the unobservable transitions. Let

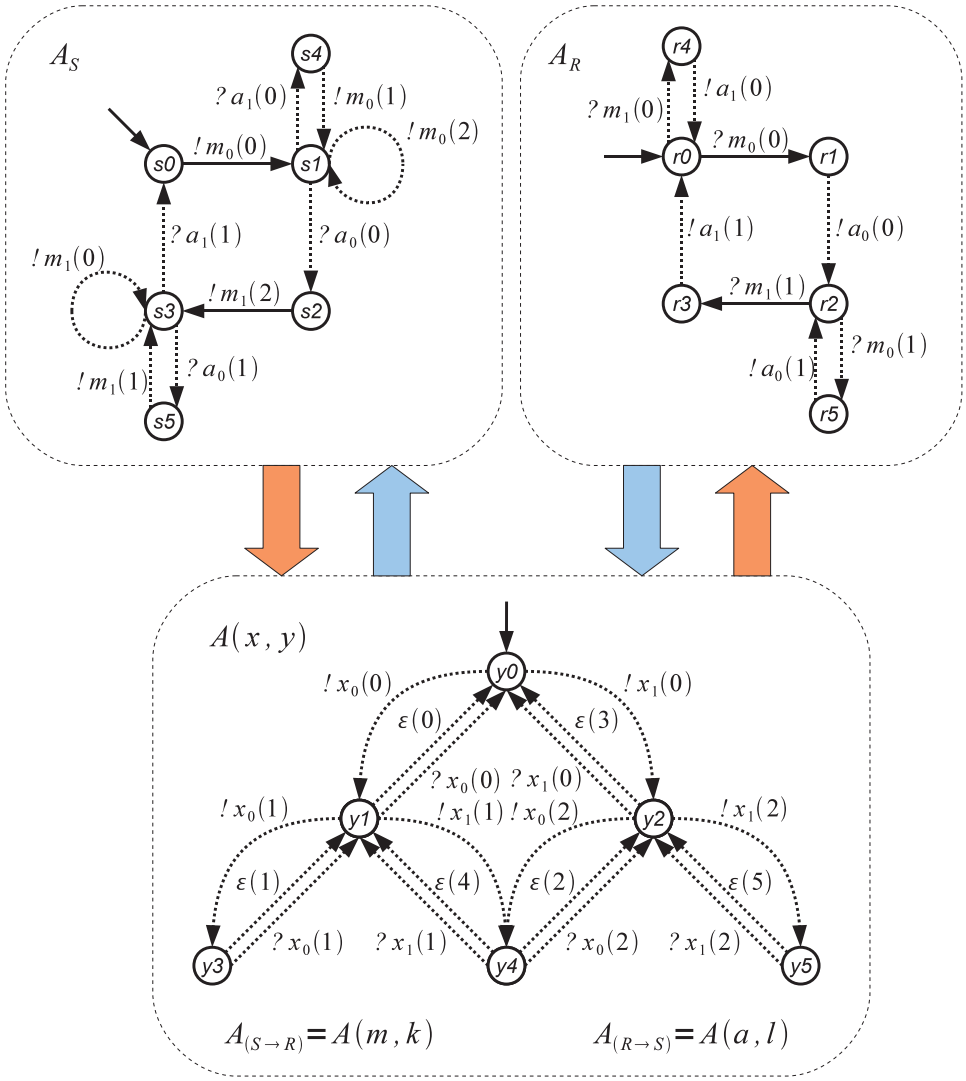


Figure 3.6: The alternating bit protocol and three different automata, each of which has its role in the protocol.

us also mention that the user do not observe any transitions of the automata which model medium.

In our example, we want to answer the following questions:

- Given the sequence of observations $\sigma = !m_0\,(0), ?m_0\,(0), ?m_1\,(1), !m_1\,(2), ?m_0\,(0)$, is there an execution of the given model which satisfies σ?

  In other words, we check if there is a possibility of sending a message once (in our example: $!m_0\,(0)$) and receiving it twice by the receiver ($?m_0\,(2)$).

- How do such possible executions look like?

- How can we avoid such executions when we take into account time constraints of the model?

To address the first and the second question, we can use the method proposed in Section 3.3 and we can construct the prefix of the unfolding of the network. Because there are some unobservable events and we want to terminate the construction of the prefix at some point, we give a limit of maximum five unobservable events in any execution (as we mentioned, the problem of unobservable events is analyzed in details in Chapter 4).

The possible explanations with respect to the given sequence of observations and the termination condition are presented in Figure 3.7. There are twelve successful executions, each with different maximal events (drawn using double ellipses). In the figure, there are only events which are successful and their causal predecessors. The rest of the events produced during the search procedure are removed. Dotted arrows are unobservable transitions, dotted ellipses unobservable events. The events drawn using double ellipses are the ones which satisfy the sequence of observations. It is worth noting that in some explanations, the final events are unobservable e.g. the event e_{305}.

Let us mention that four of these events $e_{307}, e_{308}, e_{309}, e_{310}$ are not real events. We introduced them to represent executions which end up with more than one maximal event. During construction of the prefix, the actual number of events was much higher than presented in the figure (307 events were generated for this example).

Before we answer the last question, we give below the constraints associated with the automata in Figure 3.6:

- for the sender A_S, we have the following constraints:

$$\gamma\left(!m_0\left(2\right)\right) \stackrel{def}{=} c_S \geq \beta$$
$$\gamma\left(!m_1\left(0\right)\right) \stackrel{def}{=} c_S \geq \beta$$
$$\rho\left(!m_0\left(0\right)\right) \stackrel{def}{=} \{c_S\}$$
$$\rho\left(!m_1\left(2\right)\right) \stackrel{def}{=} \{c_S\}$$

- for the two automata $A_{S\to R}$ and $A_{R\to S}$:

$$\rho\left(!x_0\left(0\right)\right) \stackrel{def}{=} \rho\left(?x_0\left(1\right)\right) \stackrel{def}{=} \rho\left(?x_0\left(2\right)\right) \stackrel{def}{=} \{c_A\}$$
$$\rho\left(!x_1\left(0\right)\right) \stackrel{def}{=} \rho\left(?x_1\left(1\right)\right) \stackrel{def}{=} \rho\left(?x_1\left(2\right)\right) \stackrel{def}{=} \{c_A\}$$
$$\rho\left(\varepsilon\left(1\right)\right) \stackrel{def}{=} \rho\left(\varepsilon\left(4\right)\right) \stackrel{def}{=} \rho\left(\varepsilon\left(2\right)\right) \stackrel{def}{=} \rho\left(\varepsilon\left(5\right)\right) \stackrel{def}{=} \{c_A\}$$
$$\forall_{i\in[1,5]} I\left(y_i\right) \stackrel{def}{=} c_A \leq \alpha$$

In the example, we assume that each of the automata has its own non-shared clock. In the time constraints above, we use two parameters α and β. With the parameter α, we can control the time which is spent to transmit a message through the medium (the two automata $A_{S\to R}$ and $A_{R\to S}$), whereas

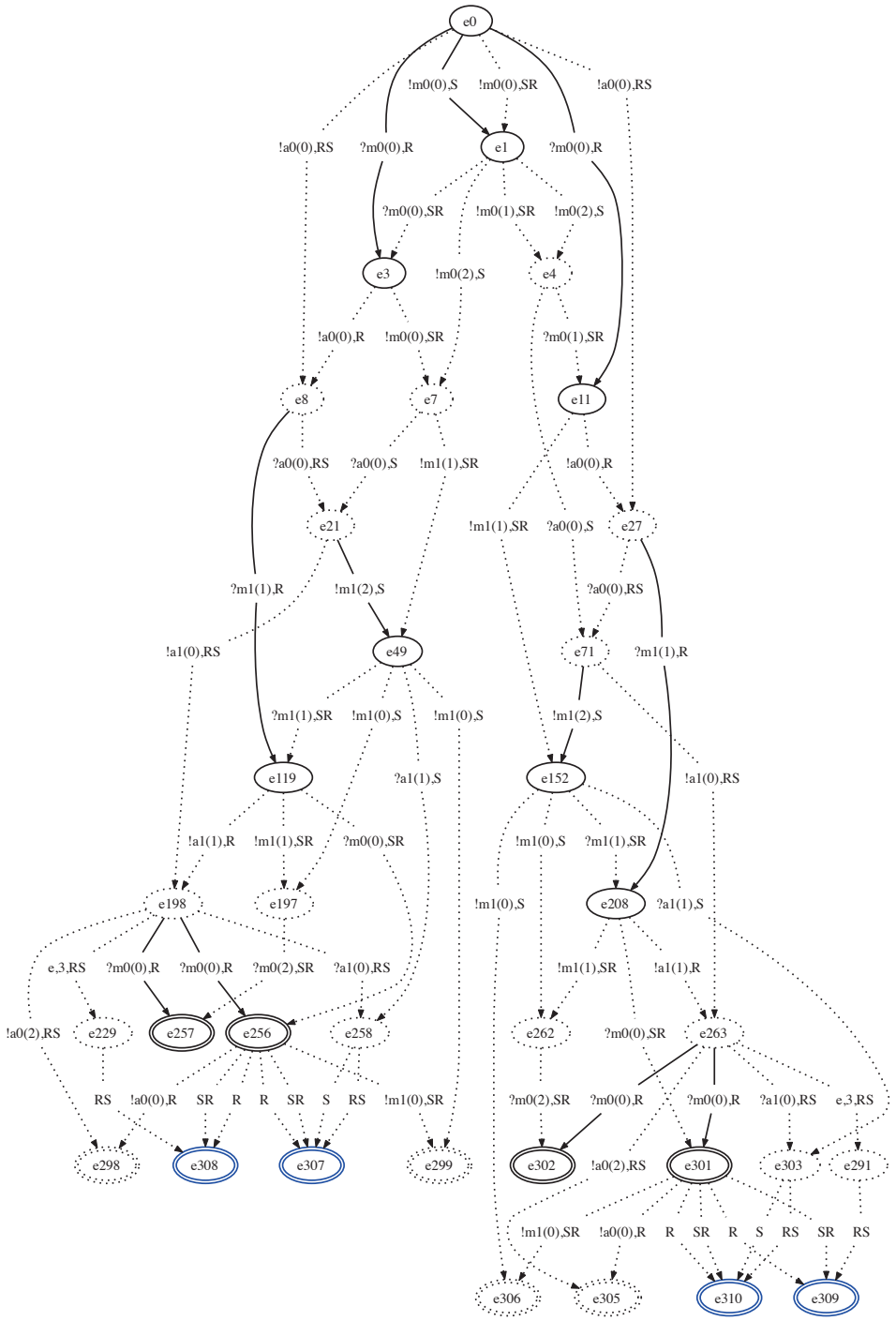


Figure 3.7: The prefix of the unfolding of the model in 3.6 constructed in accordance with the described search criteria.

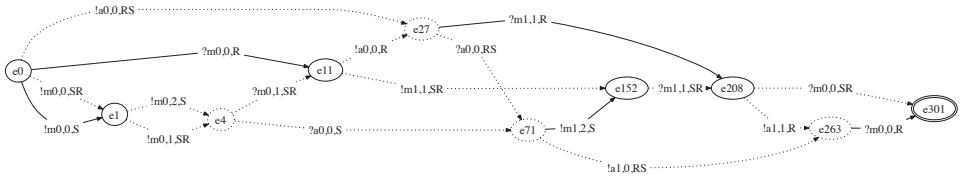


Figure 3.8: A possible explanation.

β is a threshold value of the minimal time amount after which a message can be retransmitted by the sender.

To present our method, we show on one of the successful explanations how we can avoid the corresponding execution just by adjusting the two variables α and β. For the following example, we choose the execution E shown in Figure 3.7b ending with the event e_{301}. Using Definition 33, we get the following system of constraints for this execution:

$$\begin{cases} \delta(\bot) \leq \delta(e_1) \wedge \delta(e_1) \leq \delta(e_4) \\ \delta(\bot) \leq \delta(e_{11}) \wedge \delta(e_4) \leq \delta(e_{11}) \\ \delta(\bot) \leq \delta(e_{27}) \wedge \delta(e_{11}) \leq \delta(e_{27}) \\ \delta(e_{27}) \leq \delta(e_{71}) \wedge \delta(e_4) \leq \delta(e_{71}) \\ \delta(e_{11}) \leq \delta(e_{152}) \wedge \delta(e_{71}) \leq \delta(e_{152}) \\ \delta(e_{27}) \leq \delta(e_{208}) \wedge \delta(e_{152}) \leq \delta(e_{208}) \\ \delta(e_{208}) \leq \delta(e_{263}) \wedge \delta(e_{71}) \leq \delta(e_{263}) \\ \delta(e_{208}) \leq \delta(e_{301}) \wedge \delta(e_{263}) \leq \delta(e_{301}) \\ \delta(e_{301}) - \delta(e_{208}) \leq \alpha \wedge \delta(e_{208}) - \delta(e_{11}) \leq \alpha \\ \delta(e_{71}) - \delta(e_{27}) \leq \alpha \wedge \delta(e_{152}) - \delta(e_{11}) \leq \alpha \\ \delta(e_4) - \delta(e_1) \geq \beta \wedge \delta(e_4) - \delta(e_1) \leq \alpha & (1) \wedge (2) \\ \delta(e_{11}) - \delta(e_1) \leq \alpha & (3) \\ \max_{e \in E} \delta(e) - \delta(e_{263}) \leq \alpha \end{cases}$$

When we take and reduce the three formulas (1), (2), and (3), we can simply deduce that $\beta \leq 2\alpha$. This means that it is impossible to have this execution if we set the retransmission time which is more than the double time of transmission by the medium.

3.4 Supervision using safe Petri nets

In Chapter 2, we described the notions of a place/transition net (Definition 13) and a branching process (Definition 25). Below, we introduce the basic case of supervision of Petri nets which we extend in the following sections. In this context, we also briefly discuss the problem of monotonicity of constrained unfoldings.

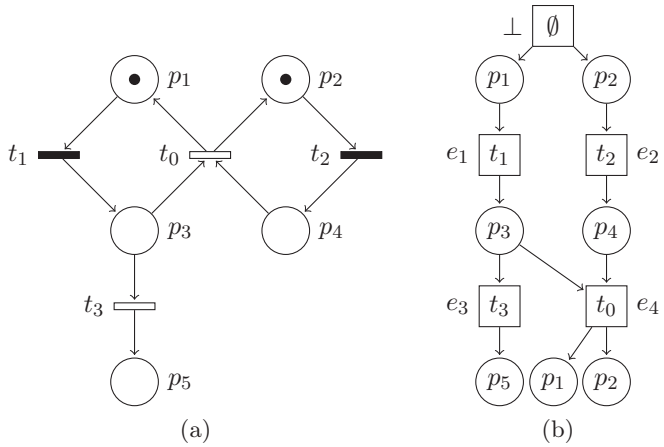


Figure 3.9: A safe Petri net (3.9a) and a constrained unfolding based on it (3.9b).

3.4.1 Constrained unfoldings of safe Petri nets

In the previous part of the chapter, we described the construction technique of constrained unfoldings of networks of timed automata. In the case of time Petri nets, we can use a similar approach.

DEF **Definition 34.** ***(Constrained unfolding of safe Petri net)*** Given a safe Petri net $\mathcal{N} = \langle P, T, W, M_0 \rangle$ and a sequence of observations σ, a *constrained unfolding* of $\mathcal{N}$, denoted by $\mathcal{E}(\mathcal{N}, \sigma)$, is the maximal branching process of $\mathcal{N}$ denoted by $\beta = \langle B, E, F, l \rangle$ such that $\forall e \in E$

- $\varsigma(\perp) = \mathbf{0}$, and
- $\varsigma(e) = \sum_{f \in \lceil e \rceil} \chi_{\lambda(\tau(f))} \leq \varpi(\sigma)$ ♦

Before we describe a short example, we introduce a simple notion of untimed time Petri net. Given a TPN $\mathcal{N} = \langle P, T, W, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$, we denote by $\mathsf{untimed}(\mathcal{N})$ the Petri net $\langle P, T, W, m_0 \rangle$.

EXM **Example 17.** In Figure 3.9a, we can see the untimed version of parametric time Petri net in Figure 2.5. We distinguish two observable transitions in the figure: t_1, t_2. We assume that both transitions are labeled with the same action. Given an observation consisting of two actions, we obtain a constrained unfolding in Figure 3.9b.

3.4.2 Non-monotonicity of constrained unfoldings

Before we directly go to discuss the problem of invisible loops, we want to draw attention to a certain property of constrained unfoldings of Petri nets. Let us consider the following example.

Example 18. In Figure 3.10a, we can see a 1-safe Petri net. In the following pictures 3.10b-3.10f, we can see constrained unfoldings of the net for 5 different observations. In the figures we left only complete explanations *i.e.* we removed events which do not belong to complete explanations. Note that each of the subsequent observations, when treated as a multiset, is a superset of the previous one, i.e. $\sigma_1 \subset \sigma_2 \subset \sigma_3 \subset \sigma_4 \subset \sigma_5$. Thus, we can somehow observe an evolution of complete explanations for the growing observation σ. Let us look closer at the event e_1 which is associated with the transition t_1. When we analyze the subsequent prefixes in Figures 3.10b-3.10f, we can observe that the event appears in Figure 3.10c, then in Figure 3.10e it disappears, and finally reappears in Figure 3.10f. EXM

In Example 18, we show that, once we want to keep only complete explanations obtained from constrained unfolding, and then want to extend the input observation of the unfolding, the resulting prefix becomes *non-monotonic*. Loosely speaking, this means that an event can be added or removed from the unfolding multiple times as the observation grows. However, as we will see in the following part of the chapter, when constructing an unfolding, we will keep all explanations, even the incomplete ones. As we have observed in the example, an event which belongs to some incomplete explanations of an observation may be a part of a complete explanation for some greater observation. Thus, in many situations, it will be computationally inefficient to remove and add the same events many times. Instead, we just keep them in the constrained unfolding during the construction, and we remove them only when there is a real need for it, *i.e.* we need to execute some particular queries on the unfolding. The above discussion can be summed up by the following lemma.

Lemma 3.1. *Let us take two constrained unfoldings $\mathcal{E}_1(\mathcal{N}, \sigma_1)$ and $\mathcal{E}_2(\mathcal{N}, \sigma_2)$, such that $\sigma_1 \subset \sigma_2$. Then, let us remove events which do not belong to any complete explanations from the both unfoldings. As a result, we respectively obtain : $\check{\mathcal{E}}_1(\mathcal{N}, \sigma_1)$, and $\check{\mathcal{E}}_2(\mathcal{N}, \sigma_2)$. The following property is true:*

$$\neg\left[\sigma_1 \subset \sigma_2 \implies \check{\mathcal{E}}_1(\mathcal{N}, \sigma_1) \sqsubset \check{\mathcal{E}}_2(\mathcal{N}, \sigma_2)\right]$$

where $A \sqsubset B$ means that A is a prefix of B (for details see Section [True conc. semantics]).

The subject of the non-monotonicity is analyzed further in Section 4.5.4, when the problem of extraction of complete explanation is presented more deeply. It is also discussed when we describe construction procedures for Petri nets.

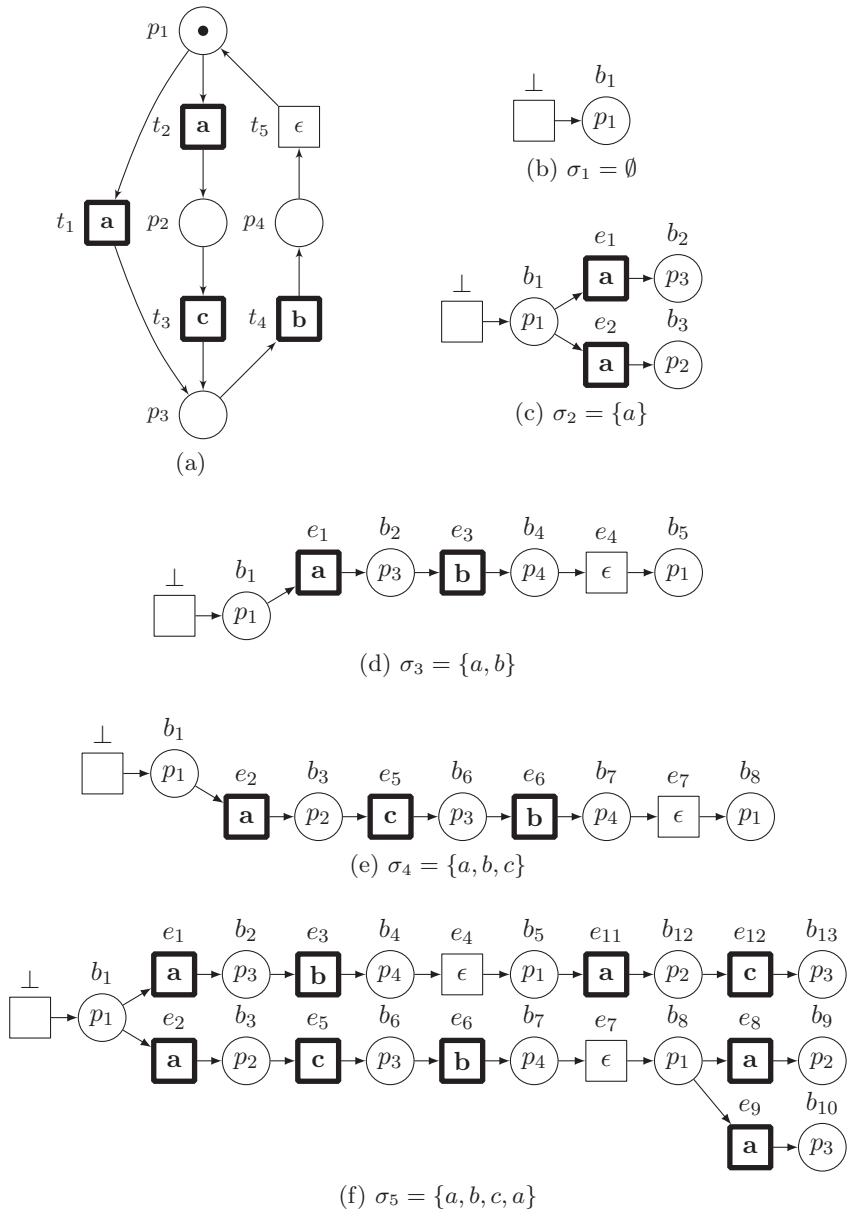


Figure 3.10: A 1-safe Petri net (a) and prefixes of constrained unfoldings for the observation σ_i ($i \in [1..5]$) only consisting of complete explanations.

3.5 Supervision with parametric time Petri nets

Below, we present the construction of a complete (*i.e.* containing all the valid explanations) constrained unfolding of a parametric time Petri net step by step for a sequence of observations σ. In the first section, we describe symbolic time branching processes which are used to construct unfoldings of PTPNs. We also present also some basic issues connected to the construction of TBPs. Next, we describe the technique which we use to unfold the parametric time Petri nets and the way to extract valid time configurations from it. In the final section, we show how to apply the unfolding method in the process of supervision. Let us note that we consider a structured observation. We terminate the presentation with a simple example of the method.

3.5.1 Symbolic time branching processes of parametric time Petri nets

Before we go into details of the unfolding technique of parametric time Petri nets we use in the process of supervision, we recall several basic notions.

We already described the semantics of parametric time Petri nets. Below, we extend this description with the most important definitions.

Definition 35. ***(Time process)*** A *time process* of a time Petri net $\mathcal{N}$ is a pair $\langle E', \theta \rangle$, where E' is a configuration of (a branching process of) $\mathsf{untimed}(\mathcal{N})$ and $\theta : E' \to \mathbb{R}_{\geq 0}$ is a timing function giving a firing date for any event of E'. ♦ DEF

Now we shall extend the notion of branching process with time information, allowing us later to define the unfolding of parametric time Petri nets. We do this in a way similar to extending configurations to time processes, by adding a function labeling events with their firing date. However, in a branching process some events may be in conflict, which means that some of them may not fire at all. We will account for this situation by labeling an event that never fires with $+\infty$.

Definition 36. ***(Time branching process)*** Given a PTPN $\mathcal{N} = \langle P, T, W, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$, a *Time Branching Process* (TBP) of $\mathcal{N}$ is a tuple $\langle \beta, v, \theta \rangle$ where $\beta = \langle B, E, F, l \rangle$ is a branching process of $\langle P, T, W, m_0 \rangle$, $v \in D_\Pi$ a valuation of the parameters and $\theta : E \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ a timing function giving a firing date for any event in E. ♦ DEF

The notion of time (branching) process is naturally related to the problem of validity of the timing function. In the sequel, we will say that a TBP is *valid* if its timing function is valid. We introduce such a valid timing function later in the chapter when describing the unfoldings of parametric time Petri nets.

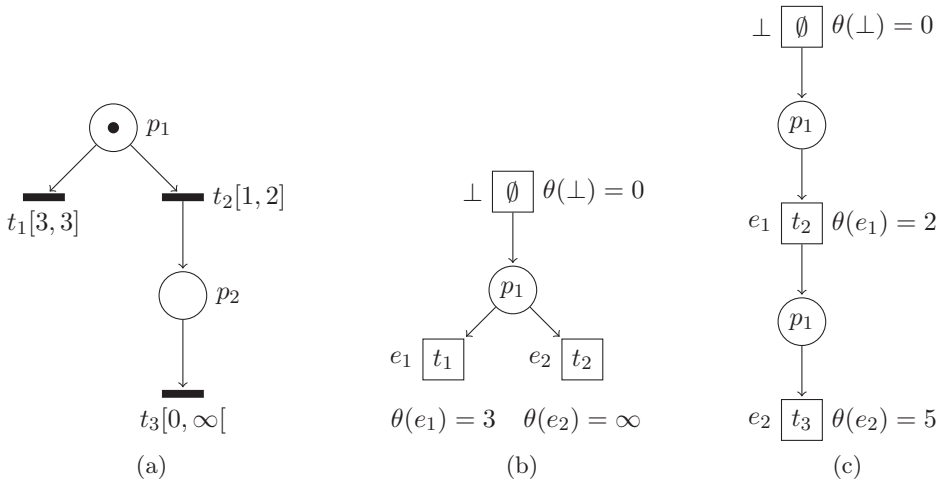


Figure 3.11: A time Petri net (3.11a) and two of its time branching processes: temporally complete (3.11b) and not temporally complete (3.11c).

Let $\mathcal{N} = \langle P, T, W, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$ be a PTPN and $\beta = \langle B, E, F, l \rangle$ the associated unfolding of $\mathsf{untimed}(\mathcal{N})$. We define the *enabling date* of an event $e \in E$ as the expression $\mathsf{TOE}(e)$ standing for $\max_{f \in {}^\bullet\!\bullet e}(\theta_f)$. It gives the date at which the corresponding transition has been enabled.

Valid time branching processes, as defined by Definitions 36, do not necessarily contain correct executions since a TBP is *a priori* incomplete in the sense that all timed constraints of the PTPN may not be included yet in the TBP: by extending the TBP with additional events, new conflicts may appear that would add those constraints. Therefore we will consider sometimes *temporally complete* TBP as defined below:

DEF **Definition 37.** ***(Temporally complete TBP)*** A TBP $\langle B, E, F, l, v, \theta \rangle$ is *temporally complete* if for all the extensions $\langle t, e \rangle$ of $\langle B, E, F, l \rangle$,

$$\max\{\theta(e') \mid e' \in E \wedge \theta(e') \neq \infty\} \leq \mathsf{TOE}(e) + v(\mathsf{lft}(t)) \tag{3.1}$$

♦

This definition basically says that the firing date of all events in the TBP should be less than or equal to the latest firing date of all possible extensions. Since the conflicts that have not yet been discovered will result from these extensions, this implies that all the events in the TBP are possible before these conflicts occur. It further ensures that all the parallel branches in the TBP have been unfolded to a same date. A similar condition applies for time processes.

EXM **Example 19.** Let us consider a time Petri net in Figure 3.11a. We can

observe that the time branching process in Figure 3.11b is temporally complete. However, the time branching process in Figure 3.11c is not temporally complete. Note that the latest firing date is 5 and that there exists a possible extension $\langle t_1, e_3\rangle$ with the latest firing date 3.

In this context, a question about the construction of a time branching process may arise. Namely, how to extend a time branching process in order to ensure that it is temporally complete? Firstly, it was shown (*e.g.* in [85]), that for a temporally complete TBP $\langle B, E, F, l, v, \theta\rangle$ and any extension $\langle t, e\rangle$ of $\beta = \langle B, E, F, l\rangle$, there exists θ' such that $\langle \beta', v, \theta'\rangle$ is a valid TBP in which β' is the branching process obtained by extending β. Secondly, if the extension of β has the smallest latest firing date of all the extensions, then $\langle \beta, v, \theta\rangle$ extended by $\langle t, e\rangle$ is a temporally complete TBP. Thus, following these two rules from the initial event $\perp$, we obtain a temporally complete TBP.

Similarly to the case of networks of timed automata, we can consider more than one timing functions in a time branching process. Having all the possible timing functions and all the possible valuations for the parameters providing such TBPs, we obtain a union of convex polyhedra in the rational space of the infinite dimension, denoted by $\mathcal{D}(\mathcal{N})$. Also, whatever the parameter valuation and the timing function, the underlying branching process of the corresponding infinite TBP is the same. We denote it by $\beta(\mathcal{N})$. In the following part we refer to $\mathcal{D}(\mathcal{N})$ (respectively $\beta(\mathcal{N})$) as to $\mathcal{D}$ (respectively β), if the argument is clear from the context.

Formally, we can express a symbolic time branching process as follows:

Definition 38. ***(Symbolic time branching process)*** Let $\mathcal{N} = \langle P, T, W, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi\rangle$ be a PTPN. A *symbolic time branching process* (STBP) Γ is a pair $\langle \beta, \mathcal{D}\rangle$ where $\beta = \langle B, E, F, l\rangle$ is a branching process of $\mathsf{untimed}(\mathcal{N})$, $\mathcal{D}$ is a subset of $\mathbb{Q}^{|\Pi|} \times (\mathbb{R} \cup \{+\infty\})^{|E|}$ such that for all $\lambda = (v_1, \ldots, v_{|\Pi|}, \theta_1, \ldots, \theta_n, \ldots) \in \mathcal{D}$, if we note $E = \{e_1, \ldots, e_n, \ldots\}$, v_λ the valuation $(v_1, \ldots, v_{|\Pi|})$ and θ_λ the timing function such that $\forall i, \theta_\lambda(e_i) = \theta_i$, then $\langle \beta, \theta_\lambda\rangle$ is a valid TBP of $\mathcal{N}_{v_\lambda}$. ♦ DEF

The set $\mathcal{D}$ can be represented as a union of pairs $\langle E_i, \mathcal{D}_i\rangle$ where E_i is a subset of the events (*e.g.* a configuration) of β and $\mathcal{D}_i$ is a rational convex polyhedron (possibly of infinite dimension) whose variables are the events in E_i plus the parameters of the net. Each point λ in $\mathcal{D}_i$ describes a value of the parameters and the finite values of the timing function on the elements of E_i. For all elements not in E_i, the timing function has value $+\infty$. We can find an example of a symbolic TBP in Figure 3.13.

As in the case of a branching process, we can define a notion of prefix of symbolic time branching processes.

Definition 39. ***(Prefix of an STBP)*** Let $\mathcal{N} = \langle P, T, W, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi\rangle$ DEF

be a PTPN. Let $\langle \beta, \mathcal{D} \rangle$ and $\langle \beta', \mathcal{D}' \rangle$ be two STBPs of $\mathcal{N}$. $\langle \beta, \mathcal{D} \rangle$ is a *prefix* of $\langle \beta', \mathcal{D}' \rangle$ if β is a prefix of β' and $\mathcal{D}$ is the projection of $\mathcal{D}'$ on the parameters plus the events of β. ⧫

There also exists the greatest symbolic time branching process according to this relation which is called the *symbolic unfolding*. What is significant, as we can observe in the next section, is that the unfolding obtained using the technique presented below has the same size as the one computed for underlying Petri net. However, some of the events may not be executed in any circumstances. These events are not *possible* and will be useless. Thus, it will be sufficient to compute a prefix of the unfolding in which they are discarded.

3.5.2 Unfolding parametric time Petri nets

The method we propose to unfold time parametric Petri nets is based on an original way of determining conflicts in the net. The method with all its details, which we shortly present below, was introduced and described in [84, 85]. In the non parametric case, unfoldings built with this method generally differ from those of [35]. In [35], the emphasis is put on the on-line characteristic of the algorithm: it is a pessimistic approach that ensures that events and constraints put in the unfolding cannot be back into question. This possibly leads to unnecessary duplication of events. In the more recent method, a more optimistic approach is applied: it requires to dynamically compute the conflicts and sometimes to backtrack on the constraints. For this purpose we use a refined version of the conflict notion, *i.e.* a relation of direct conflict.

Direct conflicts involve events sharing a common precondition but such that none of their preconditions are in conflict. The formal definition is as follows:

DEF **Definition 40.** ***(Direct conflict)*** Let $\mathcal{O} = \langle B, E, F \rangle$ be an occurrence net. Two events $e_1, e_2 \in E$ are in direct conflict, which we denote by $e_1 \mathsf{conf}\ e_2$, iff:

- ${}^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset$, and
- $\forall b \in {}^\bullet e_1, \neg(b \# e_2)$, and
- $\forall b \in {}^\bullet e_2, \neg(b \# e_1)$. ⧫

The last two conditions amount to say that ${}^\bullet e_1 \cup {}^\bullet e_2$ is a co-set. We can note that the notion of direct conflict is central to the construction of unfoldings as direct conflicts are the cause of all conflicts. In other words, for any $e_1, e_2 \in E$, $e_1 \# e_2 \Rightarrow \exists e_1' \leq e_1, \exists e_2' \leq e_2$ s.t. $e_1' \mathsf{conf}\ e_2'$.

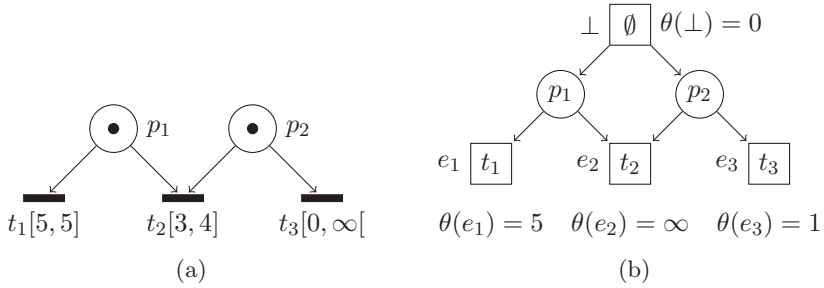


Figure 3.12: A time Petri net with conflicts (3.12a) and one of its possible time branching processes (3.12b).

Example 20. In Figure 3.12a, there is an example of a simple time Petri net with conflicts. In Figure 3.12b, we have its time branching process which we explain later in the chapter. There are the two direct conflicts: e_1 conf e_2, e_2 conf e_3. EXM

Having the notion of the direct conflict, we can proceed to unfold a parametric time Petri net. The idea is to decorate the unfolding of the underlying net by associating with each event e a symbolic expression $\theta(e)$ representing the constraints that must be satisfied to justify the occurrence of e. For each event e, we consider its firing date represented by the variable θ_e. The expressions on events will be boolean expressions on linear constraints on the set of variables and parameters. Figure 3.13 gives an example of such "decorated" unfolding.

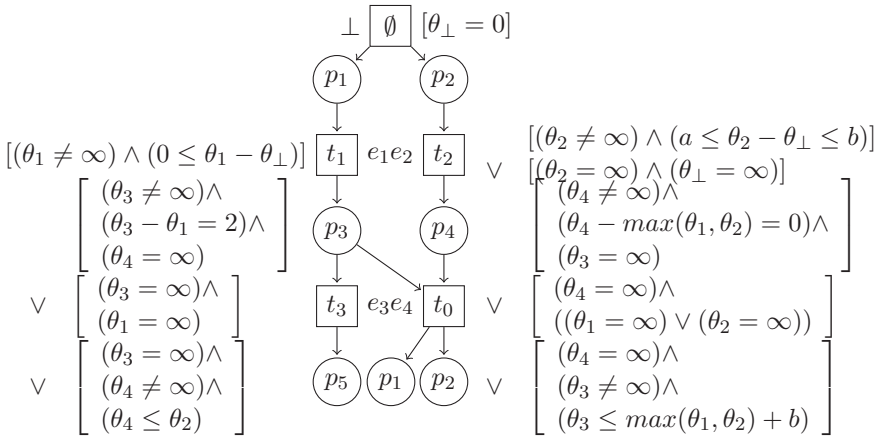


Figure 3.13: A prefix of the symbolic unfolding of the PTPN of Figure 2.5.

Definition 41. ***(Valid timing function for an unfolding)*** Given a DEF

PTPN $\mathcal{N} = \langle P, T, W, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$. Let $\beta = \langle B, E, F, l \rangle$ be the unfolding of $\mathsf{untimed}\,(\mathcal{N})$. The *timing function* θ is defined by $\theta(\bot) = 0$ and $\forall e \in E$ $(e \neq \bot)$,

$$\left[\begin{array}{lll} (\theta_e \neq \infty) & \wedge & (\mathsf{eft}\,(l\,(e)) \leq \theta_e - \mathsf{TOE}(e) \leq \mathsf{lft}\,(l\,(e))) \\ & \wedge & \bigwedge_{e' \in E, e' \text{ conf } e} (\theta_{e'} = \infty) \end{array} \right] \tag{3.2}$$

$$\vee \left[\begin{array}{lll} (\theta_e = \infty) & \wedge & \bigvee_{b \in {}^\bullet e} (\theta_{{}^\bullet b} = \infty) \end{array} \right] \tag{3.3}$$

$$\vee \left[\begin{array}{lll} (\theta_e = \infty) & \wedge & \bigvee_{e' \in E, e' \text{ conf } e} \left[\begin{array}{ll} & (\theta_{e'} \neq \infty) \\ \wedge & (\theta_{e'} \leq \mathsf{TOE}(e) + \mathsf{lft}\,(l\,(e))) \end{array} \right] \end{array} \right] \tag{3.4}$$

♦

Note that, in this definition, the parameters appear through the functions eft and lft.

The first line of the expression (Equation 3.2) means that the event e has been fired and, consequently, that no conflicting event is fired and that its firing date must conform to its time interval according to the TPN semantics. The remaining two lines consider the case in which the event e has not been fired (coded by the expression $\theta_e = \infty$). There are two possibilities: either a preceding event has not fired yet (Equations 3.3), either a conflicting event has been fired and has prevented e to occur (Equations 3.4). This latter case means that such conflicting event has fired while e was enabled.

EXM **Example 21.** Figure 3.13 shows a symbolic prefix of the unfolding of the PTPN in Figure 2.5. We can see that each event is attributed with a symbolic expression. The expressions are formed with variables denoting the firing dates of the considered event and of its neighborhood (the events that directly precede and those in conflict) and parameters. In practice, the expressions are implemented using polyhedra.

We also define the set of events temporally preceding an event $e \in E$ as: $\mathsf{Earlier}(e) = \{e' \in E \mid \theta(e') < \theta(e) \text{ is satisfiable}\}$.

Following the standard semantics of TPNs, [8] has defined the notion of valid time configuration, which can be used here:

DEF **Definition 42.** ***(Valid time configuration)*** Let E' be a configuration of $\mathcal{U}\,(\mathsf{untimed}\,(\mathcal{N}))$. E' is a *valid time configuration* of $\mathcal{U}(\mathcal{N})$ iff $(\theta_\bot = 0)$ and

$$\bigwedge_{e \in E' \setminus \{\bot\}} \left[\begin{array}{ll} & \theta_e \geq \mathsf{TOE}(e) + \mathsf{eft}(l(e)) \\ \wedge & \bigwedge_{e' \in \mathsf{enabled}(l(\mathsf{Cut}(\mathsf{Earlier}(e)))} \theta_e \leq \mathsf{TOE}(e') + \mathsf{lft}(l(e')) \end{array} \right]$$

♦

Let us consider a maximal (in term of set inclusion) configuration E' of $\mathcal{U}(\mathsf{untimed}(\mathcal{N}))$, extended with the events that are in direct conflict E'' and equipped with the corresponding symbolic expressions of $\mathcal{U}(\mathcal{N})$. Assuming that events in E' have fired and that events in E'' not, E' is a valid time configuration if the conjunction of all expressions of $E' \cup E''$ is satisfiable. This leads to the following theorem [84].

Theorem 1. (Correctness) Let $\langle B, E, F, l, v, \theta\rangle$ be the unfolding of a parametric time Petri net $\mathcal{N} = \langle P, T, W, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi\rangle$. Consider a maximal configuration $E' \subseteq E$, and $E'' = \{e \in E \mid \exists e' \in E', e \;\mathsf{conf}\; e'\}$ THM

E' is a valid time configuration iff

$$\left[\bigwedge_{e\in E'} (\theta_e \neq \infty) \wedge \bigwedge_{e\in E''} (\theta_e = \infty)\right] \Rightarrow \bigwedge_{e\in E'\cup E''} \theta_e \text{ is satisfiable.}$$

♦

Moreover, it was proven ([84]) that all valid time processes can be represented by such a TBP. Therefore, since the symbolic unfolding contains all the valid TBPs, it also contains all the time processes of the PTPN.

Theorem 2. (Completeness) Let $\mathcal{N} = \langle P, T, W, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi\rangle$ be a PTPN and $v \in D_\Pi$ be a valuation of the parameters. Let $\langle B, E, F, l\rangle$ be a branching process of the underlying Petri net and $\langle E, \theta\rangle$ be a time process of the TPN $\mathcal{N}_\nu$. THM

There exists a temporally complete time branching process $\langle B', E', F', l', v, \theta'\rangle$ such that $\forall e \in E, \exists e' \in E'$ s.t. $l(e) = l'(e')$ and $\theta(e) = \theta'(e')$.

3.5.3 Application to supervision

Using the unfolding technique of parametric time Petri nets presented above and the notion of structured observation (see Definition 28), we can construct the constrained unfolding, *i.e.* an unfolding compatible with the observation. To define this notion of compatibility, we consider the maximal configurations and ask if they do not contain events and relations that contradict the observation. Given an observation O, we consider the Parikh vector $\varpi(O) = (|\lambda^{-1}(a)|)_{a\in\Sigma}$ which counts the number of occurrences of each action in O. The same function can also be applied to configurations, considering that, for each event e, $\lambda(e)$ is in fact $\lambda(l(e))$.

Definition 43. ***(Compatibility)*** The unfolding of a PTPN $\mathcal{N}$ is *compatible* with an observation O if all its maximal (in the sense of set inclusion) configurations are. A configuration E is *compatible* with an observation iff: DEF

1. $\forall e \in E, \varpi(E) = \varpi(O)$ and

2. $\forall o_1, o_2 \in O\,,\ o_1 \preceq o_2\ \Rightarrow$
$\exists e_1, e_2 \in E\,,\ [\lambda(o_1) = \lambda(e_1)] \wedge [\lambda(o_2) = \lambda(e_2)] \wedge (e_1 \leq e_2)$ and

3. $\forall o_1, o_2 \in O\,,\ o_1\ co\ o_2\ \Rightarrow$
$\exists e_1, e_2 \in E\,,\ [\lambda(o_1) = \lambda(e_1)] \wedge [\lambda(o_2) = \lambda(e_2)] \wedge (e_1\ co\ e_2)$ ♦

The first condition ensures that all the events in E have their counterparts in O, *i.e.* the number of all types of events has to match. The other two conditions ensure that the causal relations between some events in the observation O are satisfied in E.

THM **Theorem 3. (Finiteness)** Given a finite observation, if the PTPN does not contain loops of ϵ transitions, the set of compatible configurations is finite and thus the unfolding.

Proof. Because of the finiteness of the original Petri net, the only possibility to obtain an infinite object is to have an infinite configuration. Such a configuration contains some observable events (events e such that $\lambda(l(e)) \neq \epsilon$). They are in finite number, due to the finiteness of the observation and by application of the Parikh constraint. Thus, the only possibility is to have an infinite number of ϵ events. Because of the safeness of the net, this infinite set of events must form a chain of causality, which is prevented by the absence of ϵ-loop in the net. □

At the end of the observation, we obtain a finite unfolding in which each event is equipped with a symbolic expression. From Theorem 1, it is possible to extract the valid timed configurations. This is done by considering the maximal configurations of the underlying untimed net, extended by the events that are in direct conflict with some events of the configuration. The associated symbolic constraint is given by Theorem 1. After boolean simplification, keeping only the configurations in which the expression is satisfiable, we obtain a set of timed configuration which constitutes the set of explanations. In general an explanation adds in general a lot of information to the observation:

- it has inferred some added causal and concurrent relations between the observable events;
- it has also inserted some patterns of non observable events;
- it gives the constraint that must be satisfied between all the firing dates of the events;
- it gives some constraints about the possible values of the parameters.

This is illustrated in Figure 3.14. We have considered the PTPN of Figure 2.5a in which only transitions t_1 and t_2 are observable and labelled with

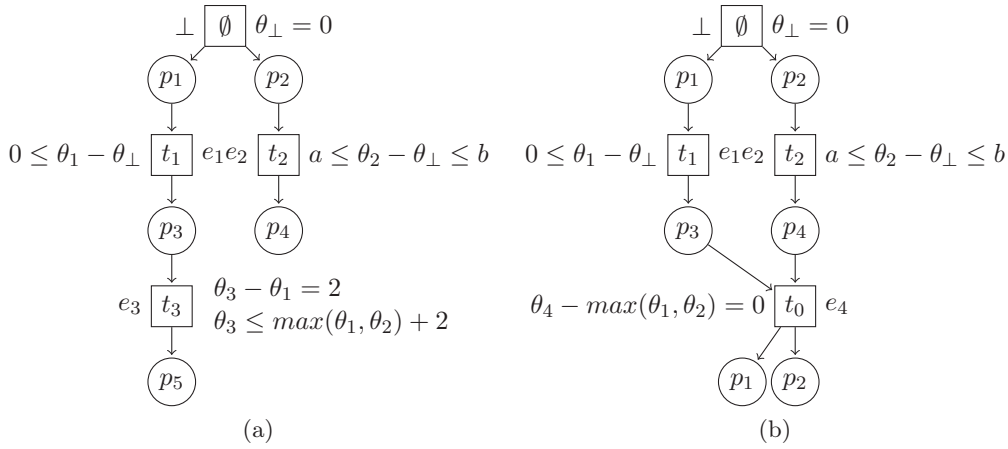


Figure 3.14: Two possible explanations.

the same letter. Now let us consider a simple observation formed with only two occurrences of the letter. The finite symbolic unfolding we obtain is the one depicted in Figure 3.13. There are only two maximal valid timed configurations as shown in Figure 3.14.

3.5.4 Case study 1

Before we go to our main case study, we briefly mention an example that directly relates to our previous example in section 3.3.2. The following case study considers a parametric model of the alternating bit protocol, but this time for parametric time Petri net. The exact analysis of the example was presented in [84]. Below, we briefly mention a few most important facts.

Like in the previous case study, the parametric time Petri net consists of four components, *i.e.* a sender, a receiver and communication channels. The sender emits messages, either with a bit (0) or (1) (transitions $!m$). After the first emission, the next retransmission happens after *to* time units, where *to* is a time parameter. In the meantime, it waits for the acknowledgment (transitions $?a$). Similarly, the receiver gets the message through transitions $?m$ and sends the acknowledgment by transitions $!a$. These two components are connected by communication channels, modeled by a 2-length queue, one for each type of messages and acknowledgments. Transmission delays belong to the parametric time intervals: $[m, M]$ for the channel of the messages, and $[a, A]$ for the channel of the acknowledgments. Messages may also be lost. To sum up, this net is safe by construction. It consists of 31 places, 29 transitions and 5 time parameters (*to*, m, M, a, A). The domain of the parameters is $D_\Pi = \{0 < a \le A,\ 0 < m \le M,\ 0 < to\}$. To supervise the

system with this model, the observable transitions of the net are labeled by actions. These are: the emissions of the sender and the correct receptions of the receiver. All other transitions are considered as non-observable.

Diagnostic

Analogically to the case study with the network of timed automata, a sequence σ of partial observations is analyzed to determine if it is possible to send a message once and to receive it twice. For this purpose, the previously presented method of supervision is applied. First, the supervisor $\mathcal{E}(\mathcal{N}, \sigma)$ is computed by unfolding the parametric time Petri net. It contains 125 possible events and 212 conditions. Then, 43 explanations compatible with the observations are extracted from this supervisor.

For each explanation, we get a constraint on the parameters. In particular, the tool computes the fact that $to \leq M + A$ is a necessary condition to explain the observations. This result is compatible with the one obtained in [52] and described in the previous case study (Section 3.3.2). In that paper, the authors consider stricter initial constraints on the parameters ($a = A = m = M$) and get the condition $to \leq 2a$. This result can be used to prevent the sequence of observations to happen by adding the constraints $to > M + A$ to the initial domain of the parameters.

Implementation

The case study was analyzed with the tool Romeo, in a devoted module written in C++, with 3 thousand lines of code and the help of the Parma Polyhedra Library [10] to verify the satisfaction of the temporal and parametric constraints.

The proposed method assumes that there is no loop of unobservable transitions in the model. This is the case in the example. The set of explanations was limited by selecting explanations containing only a fixed number of non-observable events. Otherwise, the set might be potentially infinite.

3.5.5 Case study 2

The continuous model

In this section, we present a realistic case-study based on the industrial case study for climate control in a cowshed proposed in [58]. The problem is to keep the temperature, humidity, CO2 and ammonia concentrations at specified levels so that the well-being of pigs is ensured. Though it would be relevant to model temperature, humidity, CO2 and ammonia concentrations, we limit ourselves to modeling only temperature. It would though be easy to include the disregarded climate parameters since the mixing dynamics are, roughly, identical.

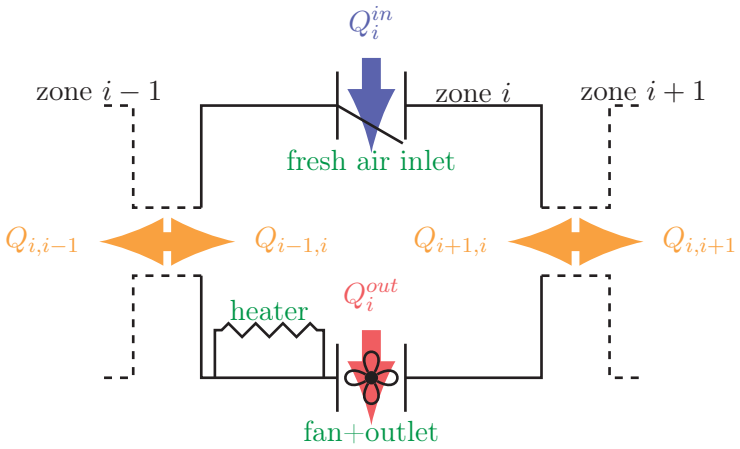


Figure 3.15: The zone number i and the air flows through it.

The cowshed is divided into distinct climatic zones which interact by exchanging air flow. Besides internal air flow, a zone interacts with the ambient environment by activating a ventilator in an exhaust pipe and also by opening a screen to let fresh air enter into the building. Air flowing from outside into the i^{th} zone is denoted $Q_i^{in}[m^3/s]$. Air flowing from the i^{th} zone to outside is denoted $Q_i^{out}[m^3/s]$. Air flowing from zone i to $i+1$ is denoted $Q_{i,i+1}[m^3/s]$(air flow is defined positive from a lower index to a higher index). A stationary flow balance for each zone i is found: $Q_{i-1,i}+Q_i^{in} = Q_{i,i+1}+Q_i^{out}$ where, by definition, $Q_{0,1} = Q_{N,N+1} = 0$. The flow balance for zone i is illustrated in Fig. 3.15

The temperature in a given zone is impacted in several ways:

- Each zone is equipped with a heater which can be either on ($u_i = 1$) or off ($u_i = 0$). We denote by $U_i[J/s]$ the resulting heating;
- The pigs in the zone produce heat, denoted by $W_i[J/s]$;
- Air flows from/to adjacent zones;
- Fresh air flows in from outside through the inlet. T_{amb} is the outside temperature. $Q_i^{in,max}$ is the maximum flow of air drawn from outside;
- Air flows outside by means of the fan. $Q_i^{out,max}$ is the maximum flow of air fanned outside.

The evolution of the temperature in zone i is therefore given by the following differential equation, where V_i is the volume of zone i, ρ_{air} the air density

[kg/m^3], and c_{air} the specific heat capacity of air [J/kg.C]:

$$\begin{array}{lcl} \frac{dT_i}{dt} & = & f(T_{i-1}, T_i, T_{i+1}), \text{with} \\ f(T_{i-1}, T_i, T_{i+1}) & = & \frac{1}{V_i}[Q_i^{in} T_{amb} - Q_i^{out} T_i + Q_{i-1,i} T_{i-1} - Q_{i,i-1} T_i \\ & & -Q_{i,i+1} T_i - Q_{i+1,i} T_{i+1} + \frac{u_i U_i + W_i}{\rho_{air} c_{air}}] \end{array}$$

Among all the factors impacting the temperature in the zone, only three ones are directly controllable:

- The heater, which is on or off;
- The aperture of the inlet, between 0 and some maximal value inducing $Q_i^{in,max}$;
- The speed of the fan, between 0 and some maximal value inducing Q^{out,max_i}.

In particular, the internal air flows between zones are induced by these last two parameters. We also decided to extend the system with an extra feature which is a possibility of failures of fans (depicted by the state OOO_i in Figure 3.16).

The parametric time Petri net generation

We consider a discrete evolution of temperature of each cell on a scale of n degrees. Each possible temperature in a cell is represented as a place. The marked place gives the current temperature of the cell. We sample and compute the successor state considering that T_{i-1}, T_i and T_{i+1} are constants. Let us denote $next_\delta(T_i)$ the temperature of cell i, obtained in these conditions after δ units of time (t.u.).

We define $C_\delta \in [0, 1]$ as the coefficient of heat exchange on the duration δ.

1. Without fan (no communication with outside) and without pig:

$$next_\delta(T_i) = T_i + C_\delta * \frac{T_{i-1} - T_i}{3} + C_\delta * \frac{T_{i+1} - T_i}{3}$$

   On an infinite delay ($C_\delta = 1$), we would obtain the heat equilibrium: $next_\delta(T_i) = \frac{T_{i-1}+T_i+T_{i+1}+T_{amb}}{3}$

2. Without fan, but with pigs:
   Let T_δ^W be the heat brought by the pigs during δ time units.

$$next_\delta(T_i) = T_i + C_\delta * \frac{T_{i-1} - T_i}{3} + C_\delta * \frac{T_{i+1} - T_i}{3} + T_\delta^W$$

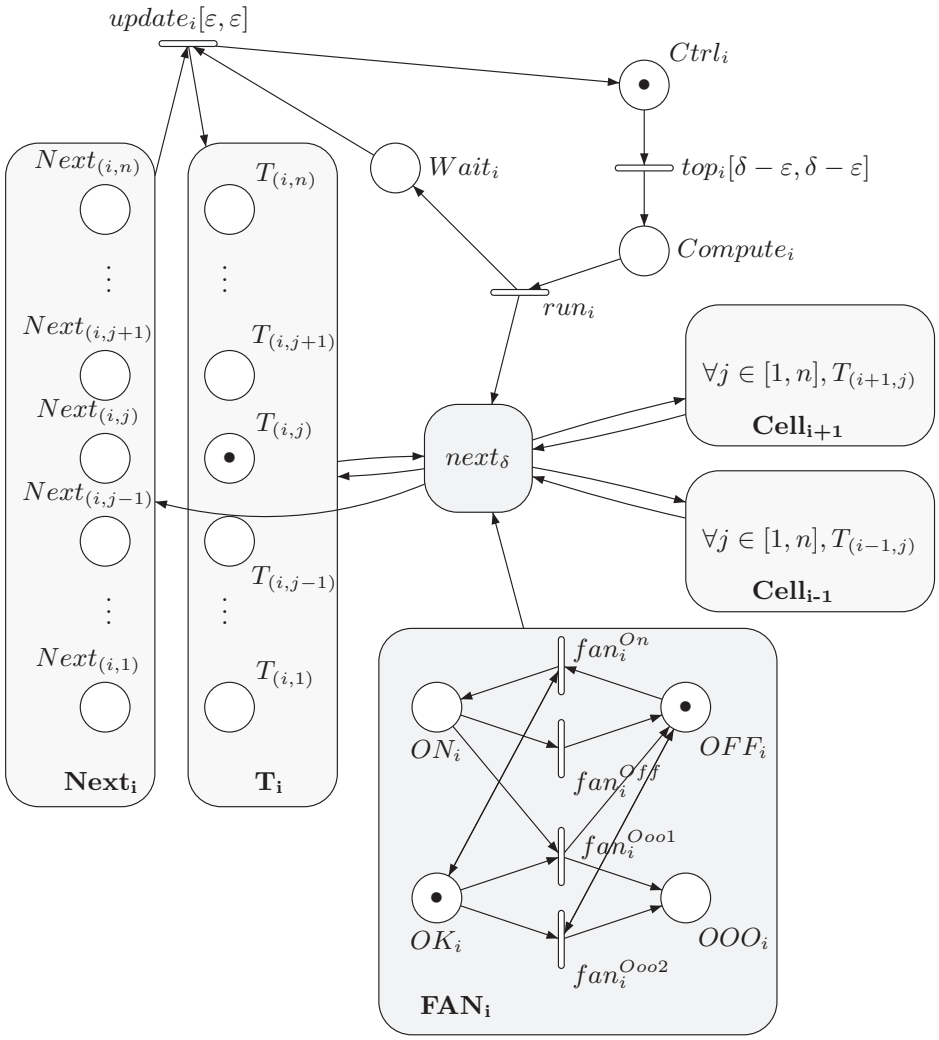


Figure 3.16: The cell i.

3. With fan and pigs:
   Let $C_\delta^{amb} \in [0,1[$ the coefficient of heat exchange with outside (depending on the power of the fan: $C_\delta^{amb} = 1$ means a fan with an infinite power).
   $next_\delta(T_i) = C_\delta^{amb} * T_{amb} + (1 - C_\delta^{amb}) * (T_i + C_\delta * \frac{T_{i-1} - T_i}{3} + C_\delta * \frac{T_{i+1} - T_i}{3} + T_\delta^W)$

The model of a cell i, given in Figure 3.16, consists of 4 blocks:

- the block $\mathbf{T_i}$ with one place per temperature,
- the block $\mathbf{Next_i}$ is used to store the intermediate state,
- the block $\mathbf{FAN_i}$ is a model of the behavior of the fan including possibility of failures,
- the block $next_\delta$ compute the next temperature of the cell and performs the exchange of tokens between blocks $\mathbf{T_i}$ and $\mathbf{Next_i}$ using the block $\mathbf{FAN_i}$ and the temperature of adjacent cells. This exchange is given by the quantization (on the n temperature levels) of the function $next_\delta(T_i)$.

The sampling is controlled by the places $Ctrl_i$ $Compute_i$ and $Wait_i$ and transitions top_i, run_i and $update_i$. The new temperature of the cell i is computed in two steps. First, the new temperature $next_\delta(T_i)$ is computed at $(\delta - \epsilon)$ t.u. and the result is stored in the intermediate places of block $\mathbf{Next_i}$. This intermediate result is obtained in zero t.u. and does not depend on the interleaving since marking of block $\mathbf{T_i}$ is not modified by this computation. Then, the intermediate result is moved from $\mathbf{Next_i}$ to $\mathbf{T_i}$ after ε t.u.

All the possibilities are defined, which leads to a complex graph. With n the number of temperature levels, the model has $2 \times n^3$ transitions. This one is tedious to build by hand. Thus, we decided to automatically generate the model by programming a tcl-tk generator, parameterized by the number of considered cells and the temperature scale. This generator of 1000 lines builds a PTPN model as an XML file directly read by Romeo. For example, Figure 3.17 gives an insight of the model with 2 cells and 3 levels of temperature.

The diagnosis experiment

The goal of the experiment is to show a case in which a certain maximal temperature is exceeded in one of the cells. This leads in turn to the death of some pigs. In the model, we assume that we can observe the changes of temperature in each of the cells, and that we can get to know if some pigs are dead. As a result, we would like to know the possible explanations of cases in which a pig died. For example, we can imagine a situation where a

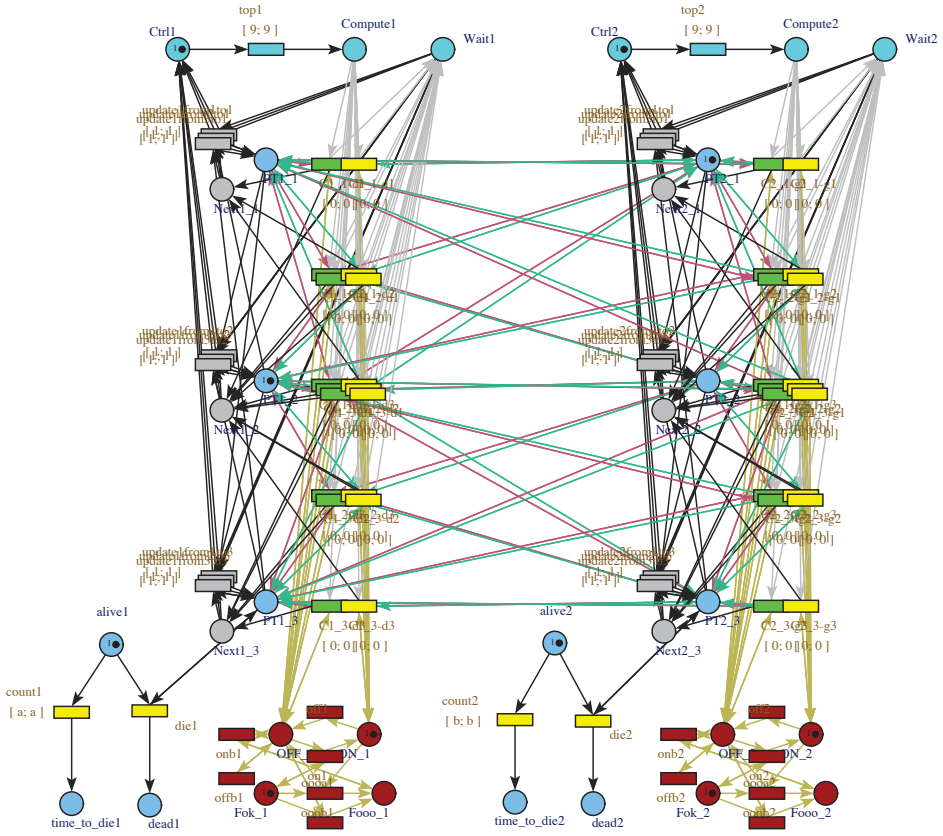


Figure 3.17: The model with 2 cells and 3 levels of temperature.

fan is broken in a cell. Moreover, we would like to obtain some information about the possible dates of death.

The experiment was performed on the system presented in the previous section in Figure 3.17. It consists of 2 cells and 3 levels of temperature. In the example, we assume that the temperature of cells is monitored at some given rate which amounts to 10 units of time.

To be able to execute our scenario, we added some additional transitions to the model (see Figure 3.17). They do not change the main functionality of the model. They are used for two reasons. The first reason is to verify whether the critical temperature was reached or not. Consequently, each time the extra transitions are fired, one can observe the death of pigs as a consequence of the excessive temperature. The second reason is to compute a minimal delay between two events: the initial event of the model (at time 0), and the potential death of pigs in a cell. The structure which implements this task is based on two transitions. One of the transitions has a parametrized constraint $[a, a]$. The transitions share an input place with a token. The token is active from the very beginning of the model activity. However, the non-parametrized transition t also depends on some other token at place p (in the model it denotes the maximal temperature). Our goal is to get to know when the place p can activate the non-parametrized transition (in the model it denotes the death of a pig). We can note that, by using the structure, we get the minimal possible date of firing t by reading the parameter a.

For the purpose of the experiment, we set up an initial temperature for each cell and we entered a set of observed events into our tool for analysis, i.e.: log with unordered temperature measurements, and an event which signals death of an animal in one of the cells. In total, there were 8 observable events and a limit of 6 unobservable events in the observation. As a result, we obtain a prefix consisting of 108 events with 4 possible explanations. From the prefix, we can observe that in any of the four scenarios, the fans in both cells have to be broken before the temperature become critical. Moreover, as a result of the experiment, we get possible valuations of the parameters given in the model. Thus, we get to know that the minimal time amount which is necessary in order to reach the state dangerous for the pigs amounts to 20 units of time.

To perform the experiment, we used a prototype implemented in Romeo which is a software for analysis of time Petri nets. The experiment was executed on a small machine with 1GB of RAM and 2GHz Intel Pentium processor. The computation time of the example needed about 15 seconds. During our experiments, we also tested some different variants of the problem: with more cells, with more levels of temperature and with different observations. In general, the size of the model and observations can strongly influence the time complexity of the diagnosis. It is not difficult to observe that one of the issues which plays a great role in the time consumption of the analysis is the number of unobservable, or indistinguishable events in the

system. During the tests we observed many difficult cases in the context of time complexity.

3.6 Final remarks

In the first part of the chapter we present an original method using the model of networks of timed automata to compute timed explanations of a sequence of actions produced by a distributed system under surveillance. Many possible applications can be considered such as the correlation of alarms and the detection of errors, monitoring behavior patterns to detect for example intrusions, surveillance of non-functional time properties, *etc.* A more detailed algorithmic analysis is given in Chapter 5. Concerning the algorithmic complexity, we can note that one of its main components is nondeterminism caused by unobservable events. The more unobservable events in the partial observation, the greater the number of possible explanations of the model execution.

During our research, we developed a prototype tool which implements the presented solutions. We used this tool to prepare a more realistic example which is described in Section 3.3.2.

In the second part of the chapter, we propose a new technique for the unfolding of safe parametric time Petri nets that allows a symbolic handling of both time and parameters. To the best of our knowledge, this is the first time that the parametric cases are addressed in the context of unfoldings. Moreover, when restricting to the subclass of safe time Petri nets, our technique compares well with the previous approach of [35]. It indeed provides a more compact unfolding, by eliminating the duplication of transitions, and also removes the need for read arcs in the unfolding. As a tradeoff, the constraints associated with the firing times of events may seem slightly more complex.

We have partly implemented the technique in the tool Romeo which can currently compute unfoldings of safe time Petri nets. However, the computation of the finite prefix is not yet implemented and the unfolding is there coupled to a supervision technique that makes the unfolding finite based on a finite set of observations.

The current version of the Romeo tool 2.9.0 is available on the webpage [2]. It also offers the possibility of computing symbolic unfoldings for safe time Petri nets with parameters. When guided by a sequence of actions, this feature allows the user to perform some diagnosis. The diagnosis consists of a finite prefix of the unfolding, presenting all the possible explanations of the input sequence. The explanations explicit the inferred causal relationships between the events of the model and also give the possible values for the parameters. We think that such an integrated method is a real added-value for the analysis of concurrent systems, and opens the door to deal with even

more complex models like time Petri nets with stopwatches or time Petri nets with more robust time semantics (e.g. with imperfect clocks).

It is worth noting that in [85] we can already find a method to unfold safe parametric stopwatch Petri nets. Stopwatch Petri nets [21] are a strict extension of the classical time Petri nets *à la* Merlin (TPNs) [72, 20] and provide a means to model the suspension and resumption of actions with a memory of the "work" done before the suspension. This is very useful to model real-time preemptive scheduling policies for example.

Chapter 4

Unobservable loops

Below, we address the problem of constructing an unfolding without having to consider the infinite behaviors of ϵ-transitions. It is motivated by the application of supervision or diagnosis.

4.1 Introduction

We propose a new method to deal with unobservable and infinite behaviors for *constrained unfoldings* of Petri nets. The concept of constrained unfolding was introduced in [46] to tackle the problem of *supervision* of distributed systems. Loosely speaking, it is assumed that such systems provide information about certain events that occur in it. Then this information is analyzed in order to reproduce the possible scenarios that could occur during the system operation.

In this chapter, we address the particular question of unfolding Petri nets under *partial observations*. To understand the concept of the unfoldings under partial observations, we can imagine a system modeled by a Petri net which produces two types of events: observable and unobservable ones. By definition, only the execution of observable events can be visible outside the system. The information about observable events, called the *observation*, is collected by a certain mechanism which we call the *collector*. We take the most general case of observation which is a set of unordered events. One of the key issues in the problem is the fact that we do not want to lose the information about the unobservable events. In many situations, it is crucial to have the information in order to be able to analyze behaviors that cannot be observed. The main goal of the supervision method is to recreate a structure on the basis of the observable events, consisting of all behaviors possible for a given observation. Such behaviors are called explanations and the structure used to store them is a prefix of the unfolding of the system.

In general, the prefix which is produced by the supervision method can be infinite. This is due to unobservable loops which can be executed in the

system. That is why it is not possible to obtain all explanations for a given observation using the method mentioned above. To cope with this problem, we show how to construct a constrained unfolding under partial observation. Then we explain why it is sufficient to consider a finite part (prefix) of it only.

We present our approach step by step showing different aspects of the problem. We also show the way to convert the final constrained prefix into a 1-safe Petri net (called a *supervisor*) which contains all the explanations. Throughout the chapter, we use the bottom-up approach to present our results. We start with the simple case of finite state machine and we finish with the general case of 1-safe Petri net. In the meantime, we discuss various aspects of our solution.

The problem of supervision was also investigated in [52, 86] in the context of parametric time Petri nets and network of timed automata. Yet, the question of invisible loops still remained open there. Since the elimination of unobservable transitions from 1-safe Petri nets is still an open and very difficult question (see *e.g.* [88]), it cannot be circumvented by a model transformation.

4.2 Background and preliminaries

4.2.1 Notations

Before we go into the problem description, we shortly introduce several necessary notations.

We define a *process* of a Petri net as a branching process without conflicts.

Let $\pi = \langle B, E, F, l\rangle$ be a process. Having the notion of extension of a branching process, we can define the *future* of a process. The future of π, denoted by $future\,(\pi)$, is a set of events which can be executed after the final state of π. Additionally, we define the future of π with an extra constraint, *i.e.* we require that the events in the future belong to a given set of events S. We denote it by $future_{|S}\,(\pi)$. In other words, $future_{|S}\,(\pi) \equiv \{e \in S \mid e \in future\,(\pi)\}$.

The set of all events which precede an event e plus e is denoted by $past\,(e)$ and is formally defined as follows: $past\,(e) \stackrel{def}{=} \{f \in E \mid f \leq e\}$.

Having a configuration C and a set of events E, we denote by $C \oplus E$ the fact that $C \cup E$ is a configuration and $C \cap E \neq \emptyset$. Moreover, we say that a configuration $C_1 = C_2 \oplus E$ is an *extension* of a configuration C_2.

In order to state that two event structures E_1, E_2 are isomorphic up to the naming of events and conditions, we use an isomorphism I_1^2.

Let E be a set of events of a branching process and $C \subseteq E$, then $futures_{|E}\,(C) = \{e \in E \setminus C \mid \nexists f \in C\,,\, f\#e\}$.

4.2.2 A finite complete prefix

Once we defined the basic notions, we mention some results about finite complete prefixes of 1-safe Petri nets. The techniques used to construct the prefix are applied to many similar problems. This is also a good starting point to solve our problem.

First, note that branching processes can be partially ordered by a *prefix relation*, denoted by $\sqsubseteq$ (we already mentioned it when describing branching processes).

Definition 44. (Prefix relation, $\sqsubseteq$) Let us take two branching processes $\beta_1 = \langle B_1, E_1, F_1, l_1\rangle$ and $\beta_2 = \langle B_2, E_2, F_2, l_2\rangle$. We say that β_2 is a *prefix* of β_1 if there exists an injection $f : B_2 \cup E_2 \to B_1 \cup E_1$, such that $a = f(b) \implies l_2(b) = l_1(a)$, and the following conditions are satisfied: DEF

- $b \in B_2 \implies f(b) \in B_1 \wedge {}^\bullet b = f^{-1}[{}^\bullet f(b)] \wedge f^{-1}[{}^\bullet f(b)] \in B_2$
- $e \in E_2 \implies \begin{cases} f(e) \in E_1 \\ \forall b \in {}^\bullet f(e) \cup f(e)^\bullet,\ f^{-1}(b) \in B_2 \\ \forall b \in {}^\bullet f(e),\ f^{-1}(b) \in {}^\bullet e \\ \forall b \in f(e)^\bullet,\ f^{-1}(b) \in e^\bullet \end{cases}$

Moreover, $\forall x, y \in B_2 \cup E_2, [f(x), f(y)] \in F_1 \implies (x, y) \in F_2$, and $\forall x \in B_2 \cup E_2, l_1(f(x)) = l_2(x)$. ♦

For example, when we consider the branching process in Figure 2.7b, denoted by $\beta_1 = \langle B, E, F, l\rangle$, and we take a similar branching process $\beta_2 = \langle B \setminus \{e_3{}^\bullet\}, E \setminus \{e_3\}, F, l\rangle$, we can observe that β_2 is a prefix of β_1, *i.e.* $\beta_2 \sqsubseteq \beta_1$.

There exists a maximal branching process according to this relation for any 1-safe Petri net $\mathcal{N}$, which is called the *unfolding* of $\mathcal{N}$, denoted by $\mathcal{U}(\mathcal{N})$. Formally, $\mathcal{U}(\mathcal{N}) \equiv \max_{\sqsubseteq}[\beta \in \mathcal{B}(\mathcal{N})]$, where $\mathcal{B}(\mathcal{N})$ is the set of all possible branching processes of $\mathcal{N}$.

It was shown in [71] and later in [42] that there exists a finite prefix of such an unfolding containing all the possible reachable states. Such a finite complete prefix can be found, for example by applying the following theorem.

Theorem 4. [71] Let $\mathcal{N}$ be a 1-safe Petri net. There exists a finite prefix $\beta_f = \langle B_f, E_f, F_f, l_f\rangle$ containing all the reachable states of $\mathcal{N}$, and THM

$$\beta_f \equiv \max_{\sqsubseteq}\{\langle B, E, F, l\rangle \sqsubseteq \mathcal{U}(\mathcal{N}) \mid \nexists e, f \in E,\ e < f \wedge \text{cut-off}(e)\},$$

where $\text{cut-off}(e) \equiv \exists e' \in E, e' < e \wedge l(\text{cut}(past(e))) = l(\text{cut}(past(e')))$.

The finite prefix presented in Theorem 4 can be minimized as shown by Esparza by using so-called *adequate order* on events (for details see [44]). In

the theorem, there is the notion of a *cut-off* event which was not presented before and which is crucial in the context of this chapter. In brief, a cut-off event guarantees that there is an event before it, which leads to the same marking of the net. Thus, anytime a cut-off event is encountered during the construction of a finite complete prefix, it is set as a maximal event of the prefix.

4.2.3 Constrained unfoldings

In this section, we briefly recall what was described in the previous chapters and what we need for the current chapter.

In the context of supervision, we consider a set of observable events described by a finite alphabet Σ of actions. With these actions, we define a function $\lambda : T \to \Sigma \cup \{\epsilon\}$ that labels each transition in the net by an action $\lambda(t)$. To handle partial observations, the unobservable transitions are labeled by a silent action ϵ. In our supervision method, we will construct a prefix of the unfolding, guided by the *observation* $\sigma \in \Sigma^*$. To achieve this, we use the Parikh function of the sequence σ defined by: $\varpi : \Sigma^* \to \mathbb{N}^{|\Sigma|}$. It computes a Parikh vector $\varpi(\sigma)$ in which we count the number of occurrences of each action in σ. For each event e in a branching process $\langle B, E, F, l\rangle$, we also compute a Parikh vector $\zeta(e) \in \mathbb{N}^{|\Sigma|}$ which counts the number of occurrences of each action in the causal past of e. We can compute $\zeta(e)$ by defining for each action $a \in \Sigma$ a Parikh vector χ_a, such that all its components are set to 0, except the one corresponding to the action a which is equal to 1. χ_ϵ is the null vector. Then $\zeta(e) = \sum_{e' \in past(e)} \chi_{\lambda(l(e'))}$.

In the further part of the chapter, we use an extended version of the function ζ, denoted by $\widehat{\zeta}(E)$. $\widehat{\zeta}(E)$ is defined as follows: $\widehat{\zeta}(E) = \sum_{e' \in E} \chi_{\lambda(l(e'))}$.

DEF **Definition 45. (Constrained unfolding)** The *constrained unfolding* $\mathcal{E}(\mathcal{N}, \sigma)$ of a system modeled by a Petri net $\mathcal{N}$, for a sequence of observations σ, is a prefix β of the unfolding $\mathcal{U}(\mathcal{N})$. It is computed such that, for each event e, $\zeta(e) \leq \varpi(\sigma)$. ♦

Having the constrained unfolding, we can extract all the explanations from it.

DEF **Definition 46. (Explanation)** An explanation $\eta = \langle B, E, F, l\rangle$ is a process of a Petri net which satisfies a certain observation σ, *i.e.* $\widehat{\zeta}(E) \leq \varpi(\sigma)$. If $\widehat{\zeta}(E) = \varpi(\sigma)$, it means that η is a *complete explanation* of σ for $\mathcal{N}$. ♦

Unfortunately, the finiteness of the observation sequence does not imply that the supervisor is bounded. It remains infinite when there exist ϵ-loops in the model, that is loops containing ϵ-transitions only. Of course, this would imply the infinite number of explanations. We address this question in the following part of the chapter.

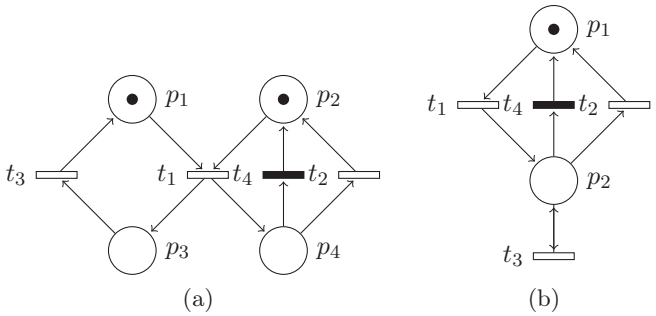


Figure 4.1: Nets with unbounded supervisors.

Example 22. In Figure 4.1, there are two Petri nets for which there is no finite constrained unfolding. Since there are unobservable loops in both nets which are available from the initial markings (the ϵ-transitions are filled in white), there is always an infinite number of events in constrained unfoldings independently of observations. EXM

In the chapter, we will also use the notion of *a supervisor* for a given Petri net $\mathcal{N}$ and observation σ, which intuitively is a Petri net such that its unfolding contains all processes of the constrained unfolding $\mathcal{E}(\mathcal{N}, \sigma)$.

4.2.4 Fusion of conditions or events in Petri nets

The fusion of places (respectively transitions) is one of the most common transformation used in the domain of Petri nets. The principal of the operation for both cases is presented in Figure 4.2. This type of operation is frequently used to reduce the size of a Petri net.

Remark that we can use a similar operation for unfoldings as they store references to original places inside conditions and references to original transitions inside events. For the purpose of the problem presented in the section, we only consider fusion of conditions (or events) which are associated with the same place (respectively transition) in the original net.

4.3 The simple case of finite state machine

4.3.1 Construction of constrained unfolding

First, we illustrate the problem and its solution on the simple case of Finite State Machines (FSM). An FSM is a particular case of safe Petri net in which the preset and postset of each transition are bounded by one:

$$\forall t \in T, |{}^\bullet t| \leq 1 \wedge |t^\bullet| \leq 1$$

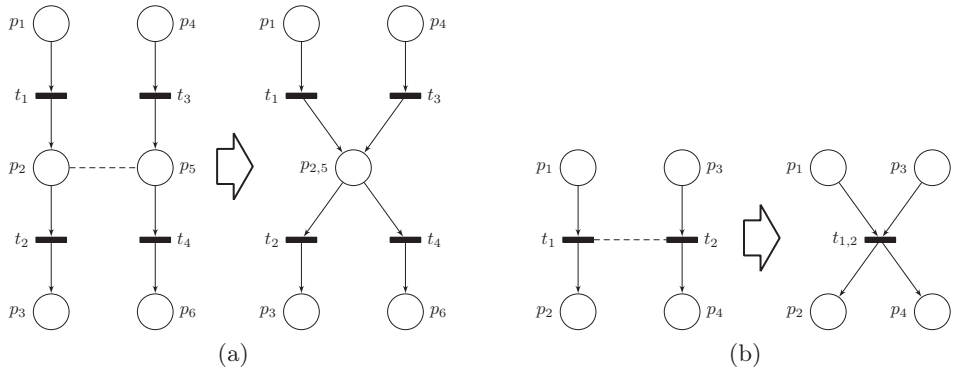


Figure 4.2: Fusion of two places (a) and two transitions (b)

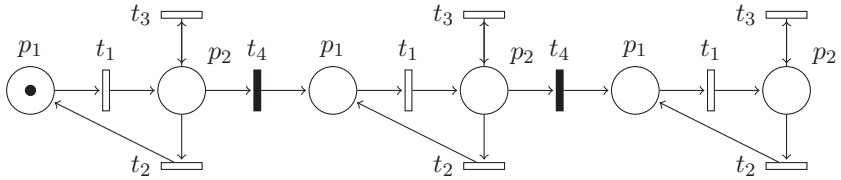


Figure 4.3: Supervisor of the net (b) of Figure 4.1 when two events are observed.

In that case, the causal relation $<$ is a total order. Thus, the branching process is just a tree branching on the conditions. Co-sets and cuts are singletons.

An *ϵ-terminal cut* is a cut such that the label of the cut (a place) is already present in the past of the condition, and such that all the intermediate events are labelled with ϵ. For each cut $C = \{b\}$,

$$\epsilon\text{-terminal}\,(C) \equiv \exists b' \in B, \begin{cases} b' < b & (1) \\ \lambda\left[l\left(b'\right)\right] = \lambda\left[l\left(b\right)\right] & (2) \\ \forall e \in E, b' < e < b \implies \lambda\left[l\left(e\right)\right] = \epsilon & (3) \end{cases}$$

b' is called the *companion* of b.

For an FSM $\mathcal{N}$ and a sequence σ of observations, let us consider the *constrained prefix* $\mathcal{P}\left(\mathcal{N}, \sigma\right)$. If the FSM contains an ϵ-loop, the constrained unfolding is infinite. We thus consider a *re-folding* operation which consists of a fusion operation of each ϵ-terminal cut with its companion. As a result, we get a Petri net $\mathcal{S}\left(\mathcal{N}, \sigma\right)$ called a supervisor.

Since the number of places is finite, the constrained prefix only contains finite ϵ-chains. Since the sequence of observations is finite too, we obtain that the constrained prefix is finite. The result is a tree (in case of non-determinism), decorated by ϵ-loops. Notice that there are no ϵ-loops with observable events in the supervisor.

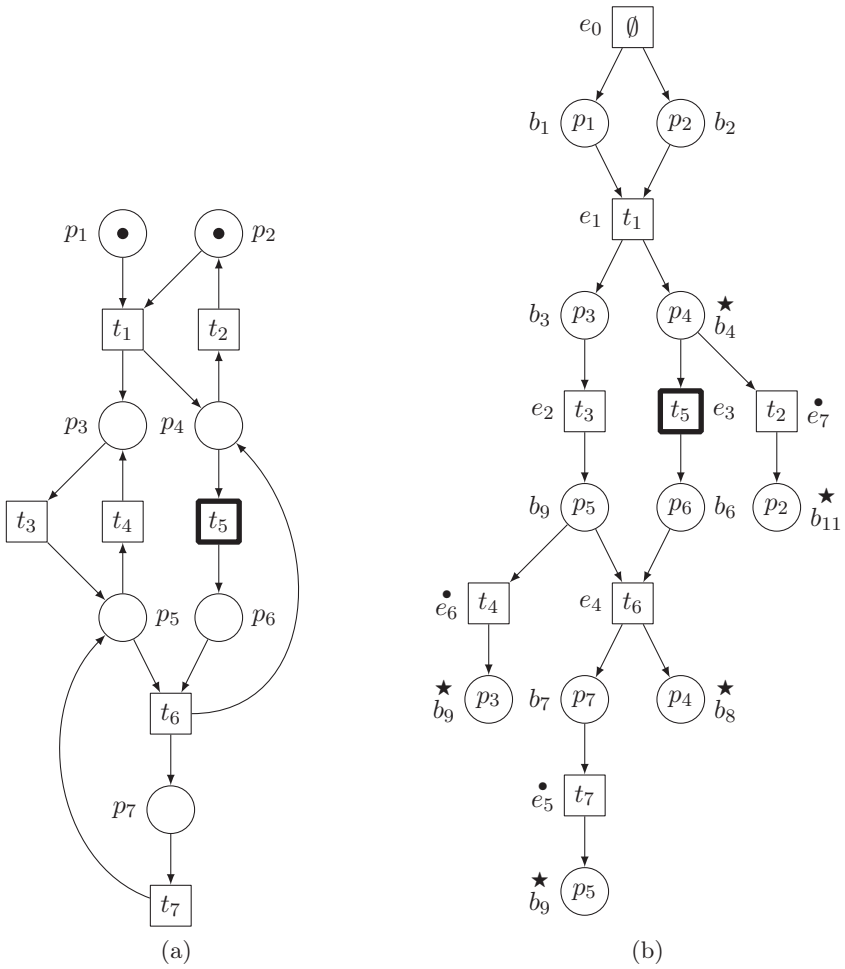


Figure 4.4: A naïve approach to solve the problem.

We can note that, in order to compute a supervisor, for each condition of the constrained prefix, we have to look if there was a condition associated with the same place in its causal past. However, as we show in Example 23, it is not true for the general case of Petri nets.

Example 23. Let us consider a Petri net in Figure 4.4a. As we can observe, it has one observable transition (drawn in bold line). In our example, we take an observation consisting of one observable event. EXM

The example shows that it does not make sense to simply glue conditions which are associated with the same places. In the example, this naïve approach leads to the fusion of two conditions, b_2 and b_{11}. Both are associated with the place p_2. However, we can see that, in order to create a loop, we would have to consider a transition t_1 which is between the two conditions.

Algorithme 4.1 Removal of incomplete explanations from constrained prefix of an FSM.

Input: A constrained prefix $\mathcal{P}$.

Output: A prefix of the constrained prefix, such that each of its events belongs to some complete explanation.

1. We take all the maximal events of $\mathcal{P}$ and we calculate their Parikh vectors.
2. If the Parikh vector of an event e is the same as the Parikh vector of the observation, we mark all the predecessors of e.
3. When we finish the analysis of all the maximal events, we remove from $\mathcal{P}$ all events which are not marked.

The rest of the conditions associated with t_1, *i.e.* b_1, b_3, does not belong to any loop in the underlying Petri net (see Figure 4.4a).

4.3.2 Removal of incomplete explanations

EXM

Example 24. In Figure 4.5a, we can see an FSM. Next to it, in Figure 4.5b, there is a Petri net which was constructed using the method from the previous section. The observation used to construct the prefix is $\sigma = \{a, b, c\}$. We can see that the structure contains some events, *i.e.* e_3, e_6, which do not belong to any *complete explanation*.

In the example 24, we show that, when we use the simple method from the previous section, we can obtain some incomplete and incorrect explanations. Such explanations can be simply removed with Algorithm 4.1.

The algorithm terminates as the number of maximal events is finite in the input prefix. Moreover, all the events in the result prefix belong to one complete explanation at least.

For the net from Example 24, the result is illustrated in Figure 4.5c.

Note that such a removal may not always be desirable, if we plan at some point to extend the observation and thus the prefix. This problem was already mentioned in the section about non-monotonicity of the supervisors (see Section 3.4.2). Recall that such incomplete explanations can become complete ones when there are some new observable events added to the observation.

4.3.3 Canonical form of the supervisor

From the previous section, we got to know how to construct a Petri net which contains all complete explanations for a given observation. However, we can go even one step further, and we can ask the question if there is a minimal

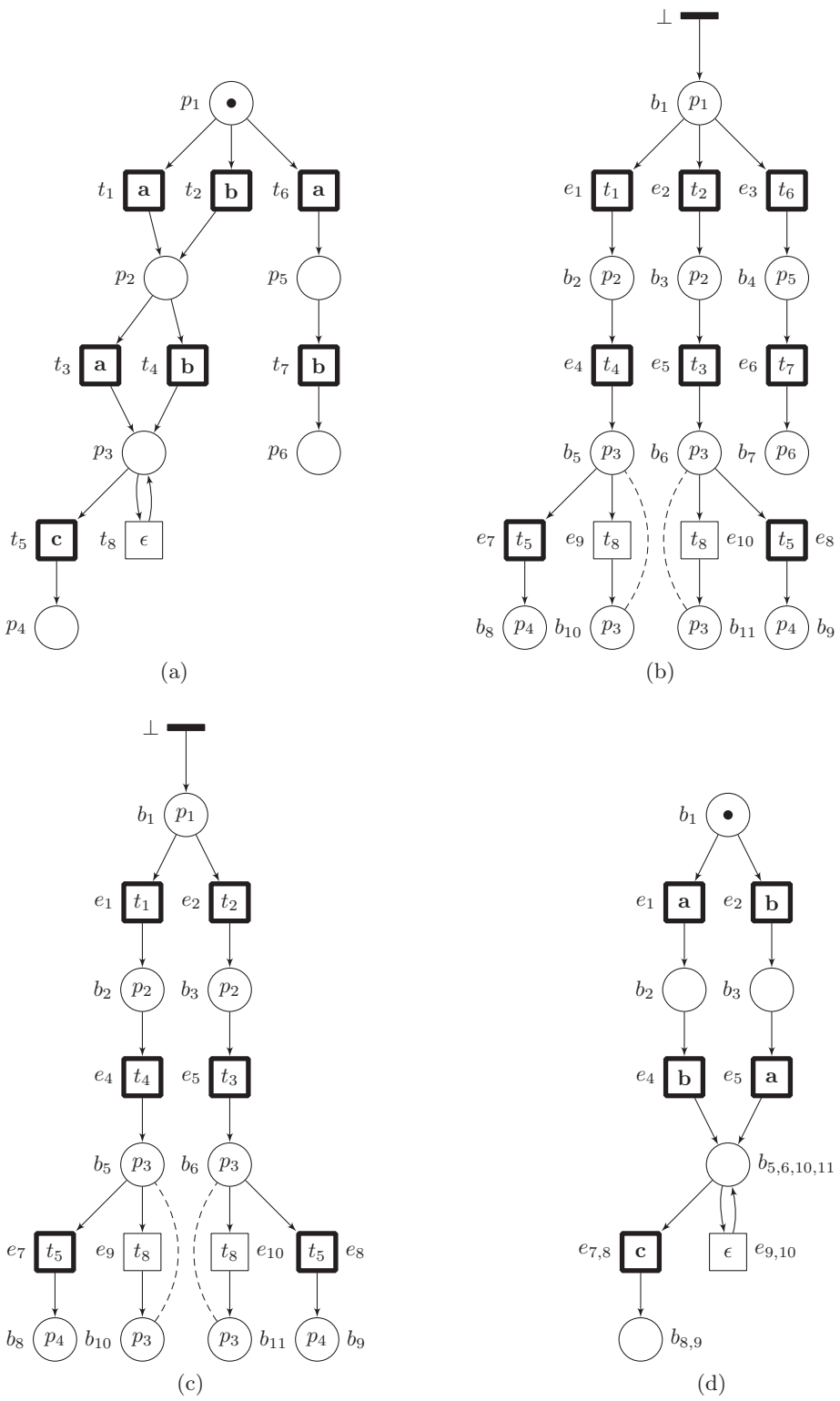


Figure 4.5: Consecutive steps to calculate canonical supervisor of an FSM.

Algorithme 4.2 Construction of canonical supervisor of an FSM.
Input: A constrained prefix with removed incomplete explanations $\mathcal{P}$.
Output: The canonical supervisor $\mathcal{S}_{norm}$.

1. Compute ϵ-state for each condition in $\mathcal{P}$.
2. Merge conditions with the same ϵ-state.
3. Merge events of E such that $\forall e_1, e_2 \in E\,,\, l\,(e_1) = l\,(e_2) \wedge \epsilon\text{-state}\,({}^{\bullet}e_1) = \epsilon\text{-state}\,({}^{\bullet}e_2) \wedge \epsilon\text{-state}\,(e_1{}^{\bullet}) = \epsilon\text{-state}\,(e_2{}^{\bullet})$,

form of such a supervisor. In other words, we ask if there is a canonical form of the supervisor. Indeed, as we will see in this section, we can compute such a form by using the constrained prefix with no incomplete explanations .

When we take a finite prefix from the previous section, we can observe that some parts of it are identical. This fact is easy to explain. Every time we have two cuts with conditions assigned to the some place , they should naturally have the same *futures*. Of course, this is true for the case when we do not consider observations in the construction of the prefix. In our case, however, we have to consider all observed events. For this purpose, let us define a state associated with each condition. Such a state, denoted $\epsilon\text{-state}\,(b)$, is a pair $[l\,(b)\,, \zeta\,(b)]$. It is simple to prove the following lemma.

Lemma 4.1. *Let us take a constrained unfolding $\mathcal{E}\,(\mathcal{N}, \sigma) = \langle B, E, F, l\rangle$ in which $\mathcal{N}$ is an FSM and σ a finite observation. For any two conditions b_1, b_2 in B, iff they have the same ϵ-states, then configurations $C_1 = past\,(b_1)$ and $C_2 = past\,(b_2)$ have the same futures.*

$$\forall b_1, b_2 \in B\,,\, \epsilon\text{-state}\,(b_1) = \epsilon\text{-state}\,(b_2) \iff futures_{|E}\,(C_1) = futures_{|E}\,(C_2)$$

Proof. The fact that the two conditions have the same ϵ-states implies the fact that they are associated with the same place in the underlying net. Moreover, the two configurations C_1, C_2 consist of the same observable actions. When we consider all the properties, and the fact that we deal with an FSM, we observe that the futures of both configurations have to be identical. □

Having the notion of the ϵ-state, we can simply compute a canonical supervisor for FSMs, denoted $\mathcal{S}_{norm}$.

Below we show that the structure computed by using Algorithm 4.2 is canonical and we explain the reason why.

THM **Theorem 5.** Given an FSM $\mathcal{N}$ and a finite observation σ, $\mathcal{S}_{norm}\,(\mathcal{N}, \sigma)$ is the smallest Petri net representing all correct explanations of σ.

Proof. There is only one place in the result net for each ϵ-state. We are also sure that each of the ϵ-states is reachable for the given observation. Thus, we cannot remove any of them. This proves that the number of places is minimal.

After the fusion of conditions associated with the same ϵ-states, we may encounter a situation in which there is more than one transition labeled by the same action such that they have identical presets. However, this also implies the identical postsets about which we are certain that they are merged in the final supervisor. This property is used in the computation of the supervisor. Thus, we are sure that there are no repetitions of such transitions. □

In fact, in implementations, we can observe that, every time we have two conditions with the same ϵ-state, we continue extending only one of them as the future will be the same for both of them. This is illustrated in Figure 4.5d.

4.4 Constrained unfoldings of free-labeled 1-safe Petri nets

There are many possible ways in which transitions of Petri nets can be labeled. A labeling function can have a significant influence on the construction procedure of constrained unfolding. In the following section, we look closer at a certain subclass of Petri nets, namely *free-labeled* Petri nets with silent transitions. The notion of free-labeled Petri net is already introduced and described *e.g.* in [77].

Definition 47. (Free-labeled Petri nets with silent transitions, FLPN) DEF
A free-labeled Petri net $\mathcal{N}_{FL} = \langle P, T, W, m_0 \rangle$ with silent transitions is a labeled Petri net in which all non-silent transitions are uniquely labeled, i.e. $\forall t_1, t_2 \in T\,,\ \lambda(t_1) = \lambda(t_2) \neq \epsilon \implies t_1 = t_2$. ♦

In the following subsections, we show how the specific properties of free-labeled Petri nets can be used in the construction of constrained unfoldings.

4.4.1 Construction of constrained unfolding

So far, we have considered finite state automata and we were able to give a procedure to construct a canonical form of constrained prefix and the associated supervisor. Below, we follow the similar steps to show the new aspects of the construction of constrained unfoldings of *FLPNs.*

First, we redefine the notion of an ϵ-terminal event.

Definition 48. **(ϵ-terminal event)** Let $\beta = \langle B, E, F, l \rangle$ be a branching DEF

process of a 1-safe free-labeled Petri net $\mathcal{N} = \langle P, T, W, m_0 \rangle$. We say that e' is an ϵ-terminal event iff the following conditions are satisfied:

$$\epsilon\text{-terminal}\left(e'\right) \equiv \exists e \in E, \begin{cases} e < e' & (1) \\ l\left[\text{cut}(past\left(e\right))\right] = l\left[\text{cut}(past\left(e'\right))\right] & (2) \\ \forall f \in \left[past\left(e'\right) \setminus past\left(e\right)\right], \lambda\left[l\left(f\right)\right] = \epsilon & (3) \end{cases}$$

Moreover, we call e an ϵ-*companion* event of e'. To express this fact, we use the function ϵ-companion $(e') = e$ or the predicate ϵ-companion (e, e'). ♦

We also use a variant of the ϵ-companion predicate but for conditions. In order to state that there are two conditions $b_1 \in \text{cut}\left(e\right)$ and $b_2 \in \text{cut}(e')$, such that $l\left(b_1\right) = l\left(b_2\right)$ and ϵ-companion (e, e') is true, we write ϵ-companion(b_1, b_2). Sometimes, we refer to cut (e) as a companion cut of cut (e').

EXM **Example 25.** In Figure 4.4, we presented a 1-safe Petri net with one observable transition. Consider Definition 48. We want to show why we cannot simply check observability of events between e and e', but we have to use the condition 3. For this reason, let us replace the last condition with the following condition: $\forall f \in E, e < f \leq e' \wedge \lambda\left[l\left(f\right)\right] = \epsilon$. When we apply the new definition of ϵ-terminal events to the net in Figure 4.4a, we will note that e_5 is an ϵ-terminal event. This would mean that transitions, which are associated with events e_3, e_4, e_5, form an invisible loop. However, t_4 is observable and such is the loop.

Given the new definition of an ϵ-terminal event, we introduce a constrained prefix for FLPN with silent transitions, which is next used to construct a supervisor.

DEF **Definition 49. (Constrained prefix)** Let $\mathcal{N}$ be a free-labeled Petri net and σ a finite observation. A constrained prefix $\mathcal{P}\left(\mathcal{N}, \sigma\right)$ is a maximal branching process such that:

$$\forall e \in \mathcal{P}\left(\mathcal{N}, \sigma\right) \begin{cases} \nexists f \in E,\ \epsilon\text{-terminal}\left(e\right) \wedge e < f \\ \zeta\left(e\right) \leq \varpi\left(e\right) \end{cases}$$

♦

As we can note, Definition 49 makes use of the basic rule of construction of constrained unfoldings, and additionally the new ϵ-terminal predicate. The similar predicate was introduced for finite state machines described in 4.3. The difference is that, this time, we may have parallel events in the configurations.

Below, we prove one of the essential properties of the constrained prefix, *i.e.* its finiteness.

Theorem 6. (Finiteness) Let $\mathcal{N}$ be a free-labeled 1-safe Petri net. Let $\mathcal{P}(\mathcal{N}, \sigma)$ be a constrained prefix. For any finite observation σ, $\mathcal{P}(\mathcal{N}, \sigma)$ is also finite, *i.e.* it consists of a finite number of events. THM

Proof. In order to prove that $\mathcal{P}$ is finite, we have to analyze two features of the structure. The first one is the maximal size of each configuration in $\mathcal{P}$. By the size of configuration, we mean the number of events in it. The second one is the number of maximal configurations which are embedded in $\mathcal{P}$. If we can show in both cases that all of those values are finite, it is equivalent with the fact that the whole structure $\mathcal{P}$ is finite.

Note that the model we consider is 1-safe. Structural features of a 1-safe Petri net make its branching process infinite only if it contains a process which is infinite. Note that any cut of any process of 1-safe Petri net is always a finite set of conditions. These properties can be directly observed when we interpret a branching process and a process of 1-safe Petri net as a graph. For a process of 1-safe Petri net, a degree of any of the vertices is always finite. Thus, we cannot have an infinite number of parallel events in the process. On the other hand, when we take a set of vertices which represents a cut of a process, it can be extended only with a finite set of events. In other words, having a state of a 1-safe Petri net, we cannot have an infinite number of conflicting events following the state.

Once we proved that there can only be a finite number of events in conflict directly extending a state of $\mathcal{P}$, we show that all the configurations are finite. We know that the observation σ is finite and that all events of $\mathcal{P}$ have to respect it. In Definition 49, this fact is represented by the condition $\zeta(e) \leq \varpi(\sigma)$. Thus, a simple conclusion is that any configuration of $\mathcal{P}$ may contain at most a finite number of observable events. Consequently, in order to make sure that the configuration is finite, we only need to guarantee that the number of unobservable events is finite. However, this situation is impossible. Let us consider a potential infinite configuration of $\mathcal{P}$, and let us take a cut C_1 of this configuration. We can note that, if the configuration is continued infinitely by executing unobservable events, eventually a cut C_2 will be reached, such that $l(C_1) = l(C_2)$. This is due to the pigeonhole principle and the fact that any 1-safe Petri net has a finite number of possible states. Thus, in such a process, we will eventually have an event which is an ϵ-terminal event. □

Now, when we have a finite constrained prefix from Definition 49, we introduce a way to obtain explanations which are coded in the structure. In order to avoid the introduction of new semantic rules, we decide to translate the constrained prefix into a Petri net. For this purpose, we introduce an operator supervisor $[\mathcal{P}(\mathcal{N}, \sigma)]$. The result of the operator is a Petri net which we call a *supervisor*. As we show in the following part, after the standard unfolding procedure, it returns a maximal branching process containing all the possible explanations of the observation σ.

Before we define a notion of a supervisor, we introduce a predicate $\epsilon\text{-}companion_{min}\,(b_1, b_2)$, where b_1, b_2 are two conditions which we use in the definition.

$$\epsilon\text{-}companion_{min}\,(b_1, b_2) \equiv \exists p = (a_1, \ldots, a_n)\,, b_1 = a_1 \wedge b_2 = a_n \wedge \\ \neg\epsilon\text{-}condition\,(a_1) \wedge \forall a_i, a_{i+1} \in p, \epsilon\text{-}companion\,(a_i, a_{i+1})$$

Note that the predicate is a subset of the transitive closure of the predicate ϵ-*companion*.

For the sake of simplicity of notation, we also introduce a predicate $\epsilon\text{-condition}\,(b) \equiv \exists e \in E\,,\ \epsilon\text{-terminal}\,(e) \wedge b \in \mathrm{cut}\,(past\,(e))$.

DEF **Definition 50. (Supervisor)** Let $\mathcal{E}\,(\mathcal{N}, \sigma) = \langle B, E, F, l\rangle$ be a constrained unfolding of a free-labeled 1-safe Petri net $\mathcal{N} = \langle P, T, W, m_0\rangle$. σ is a finite observation. We say that $\mathcal{S}\,(\mathcal{N}, \sigma) = \langle P', T', W', m'_0, l'\rangle$ is a *supervisor* based on a constrained unfolding $\mathcal{E}\,(\mathcal{N}, \sigma)$ iff:

1. $P' = \{b \in B \mid \neg\epsilon\text{-condition}\,(b)\}$
2. $T' = E \setminus \{\bot\}$
3. $\forall x, y \in B \cup E,$
$$\begin{cases} (x,y) \in W' & \text{if } \{(x,y) \in F \wedge \\ & \quad [(x \in E \wedge \neg\epsilon\text{-condition}\,(y)) \vee \\ & \quad (y \in E \wedge \neg\epsilon\text{-condition}\,(x))]\} \vee \\ & \{x \in E, \epsilon\text{-condition}\,(y) \wedge \\ & \quad \exists z \in x^\bullet, \epsilon\text{-}companion_{min}\,(y, z)\} \vee \\ & \{y \in E, \epsilon\text{-condition}\,(x) \wedge \\ & \quad \exists z \in {}^\bullet y, \epsilon\text{-}companion_{min}\,(x, z)\} \\ (x,y) \notin W' & \text{otherwise} \end{cases}$$
4. $m'_0 = \bot^\bullet$
5. $l' = l$

We express the fact of translation using the following notation: $\mathcal{S}\,(\mathcal{N}, \sigma) = \text{supervisor}\,[\mathcal{E}\,(\mathcal{N}, \sigma)]$. ♦

In fact, we can observe that the only operation used during the translation from constrained prefix to supervisor is actually the fusion of conditions described in Section 4.2.4. Apart from that, there is a small modification, *i.e.* a removal of the initial event from $\mathcal{E}$. This way, we obtain initial marking of the original net $\mathcal{N}$ in $\mathcal{S}$.

In the following part, we start with the presentation of some basic properties of the supervisor. Then, step by step, we show that the construction serves well its purpose.

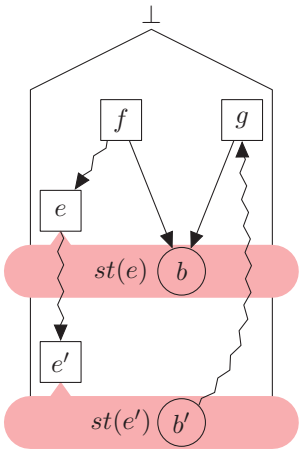


Figure 4.6: Illustration for the proof of Lemma 4.2.

Lemma 4.2. *Let us consider a process* $\pi = \langle B, E, F, l \rangle$ *of a 1-safe Petri net* $\mathcal{N}$*. Then, let us take an* ϵ-terminal *event* e' *which belongs to* π*, and its companion event* $e = \epsilon$-companion (e')*. For any condition* $b \in \text{cut}\,(past\,(e))$ *and any condition* $b' \in \text{cut}\,(past\,(e'))$*, such that* $l\,(b) = l\,(b')$*,* $b \leq b'$*.*

The intuition behind the lemma is quite simple. Namely, it states that, for every condition $b' \in cut\,(past\,(e'))$, there is a condition $b \in cut\,(past\,(e))$ assigned to the same place in the underlying net. This property is important in the context of the fusion operation which we conduct in order to construct the supervisor.

Proof. Let us assume that $l\,(b) = l\,(b')$ and $b \not\leq b'$. Because we consider a process, we have two possibilities:

1. b co b'. The situation when the two conditions are parallel and they are associated with the same place, i.e. $l\,(b) = l\,(b')$. However, we are sure that this situation is not possible. Otherwise, it would mean that there are two parallel transitions which are enabled and that they lead to the same place in the net. This would contradict the fact that we deal with 1-safe Petri net.

2. $b > b'$. From the definition of ϵ-terminal event, we know that $\exists f \in E, b \in f^{\bullet} \wedge f \in past\,(e)$. If $b > b'$, it means there is an event, let us say g, such that $b \in g^{\bullet}$ and $b' < g$. In other words, there is a sequence of events going from b' to b, and the last event in the sequence is g. This situation is depicted in Figure 4.6. The zigzag lines $\rightsquigarrow$ represent the relation $\leq$, whereas the straight lines $\rightarrow$ denote $<$. Moreover, $st\,(x) = \text{cut}\,(past\,(x))$. Now, we have two possibilities:

- $f \neq g$, this case is not possible in a branching process of 1-safe Petri net as we would have two parallel events leading to the same condition;
- $f \equiv g$, this implies the following dependencies $b' < f \leq e < e'$. This would mean that $b' \in past\,(e)$. However, we know that $b' \in \text{cut}\,(past\,(e'))$. Such a situation is not possible in general. This is due to the following property: $\forall c \in \text{cut}\,(past\,(h))\,, c \notin past\,(h)$, where h is an event, and c is a condition. The property is a direct consequence of the definitions of the functions cut and *past*.

□

Corollary 4.3. *Given a process π and two events e and e' from Lemma 4.2, let us define a sequence of branching processes $\vec{\pi} = (\pi_0, \ldots, \pi_n)$, such that:*

$\pi_0 = past\,(e)$, $\pi_1 = past\,(e')$, $\pi_i = \pi_{i-1} \oplus I\,(past\,(e') \setminus past\,(e))$. Note that when we apply Lemma 4.2 to $\vec{\pi}$, we obtain the following result:

$$\forall i, j \in [0, n]\,, \forall b_a \in \text{cut}\,(E_i)\,, b_b \in \text{cut}\,(E_j)\ ,\ i < j \wedge l\,(b_a) = l\,(b_b) \implies b_a \leq b_b$$

The intuition behind the above corollary of Lemma 4.2 is to enable us to apply the lemma to processes which are created due to a number of repetitions of an unobservable loop. This property is especially useful, once we want to prove the compatibility of the constrained unfolding with the unfolding of the supervisor (see Definition 50).

Lemma 4.4. *If $\mathcal{S}\,(\mathcal{N}, \sigma)$ is a supervisor from Definition 50, then it is a 1-safe Petri net.*

Proof. In order to break the safety of the supervisor, it would have to produce more than one token for a place. To show that it is not possible, we can take any two conditions b_1 and b_2 of the underlying constrained prefix $\mathcal{E}\,(\mathcal{N}, \sigma)$ such that they satisfy the conditions for being merged in $\mathcal{S}\,(\mathcal{N}, \sigma)$. Then, note that, in $\mathcal{E}\,(\mathcal{N}, \sigma)$, it is not possible for both conditions to be active at the same time. This is due to Lemma 4.2 and Definition 50 which show that one condition has to causally precede another. Thus, when we make a fusion of conditions of $\mathcal{E}\,(\mathcal{N}, \sigma)$, the result structure $\mathcal{S}\,(\mathcal{N}, \sigma)$ is still 1-safe. □

Before we step further into the presentation of properties of the supervisor introduced in this section, we recall the fact that we deal with free-labeled Petri nets with silent transitions. As we have already mentioned before, it is a special case of 1-safe Petri net in which each transition t, such that $\lambda\,(t) \neq \epsilon$, is labeled with a distinct action chosen from a given alphabet of actions.

In the following part, we show that, the conditions (see Definitions 45 and 51) necessary to build a supervisor can be simplified for this particular

sort of Petri nets. Namely, every time we add a new event, it is sufficient to check only the events which have the same action symbol as the new event.

We start with the following crucial remark.

Lemma 4.5. *For any two events which belong to the unfolding* $\mathcal{U}(\mathcal{N}) = \langle B, E, F, l\rangle$ *of 1-safe FLPN* $\mathcal{N}$*, if they are not in conflict and they are labeled by the same action different from* ϵ*, then they are totally ordered. Formally:*

$$\forall e, f \in E\,,\, \neg\,(e \# f) \wedge \lambda\,[l\,(e)] = \lambda\,[l\,(f)] \wedge \lambda\,[l\,(e)] \neq \epsilon \implies \neg\,(e\ \mathsf{co}\ f)$$

Proof. The explanation of the lemma is straightforward and follows from the fact that each transition of FLPN is uniquely identified by a different action, provided that the action is not equal to ϵ. Thus, two observable events which have the same label and belong to the same process cannot be executed in parallel. Otherwise it would mean that the underlying transition can be executed in parallel to itself. □

Having Lemma 4.5, we can show that the construction procedure presented in Definition 45 can be simplified. A new and more efficient procedure is presented in Definition 51.

Definition 51. (Constrained unfolding of FLPN) A constrained unfolding of FLPN $\mathcal{N}$ guided by the observation σ, denoted by $\mathcal{E}_{FL}(\mathcal{N}, \sigma) = \langle B_1, E_1, F_1, l_1\rangle$, is a prefix of $\mathcal{U}(\mathcal{N}) = \langle B_2, E_2, F_2, l_2\rangle$, such that: DEF

$$\forall e \in E_2\,,\, \widehat{\zeta}\,(e) \leq \varpi\,(\sigma)\,[\lambda\,(l\,(e))] \iff e \in E_1,$$

where $\varpi\,(\sigma)\,[\alpha]$ denotes a component of $\varpi\,(\sigma)$ associated with actions labeled by α, whereas:

$$\widehat{\zeta}\,(e) = \left|\left\{e' \,\middle|\, e' \in \mathit{past}\,(e) \wedge \lambda\,[l\,(e)] = \lambda\left[l\left(e'\right)\right]\right\}\right|.$$

♦

The following Lemma 4.6 proves the correctness of the construction presented in Definition 51.

Lemma 4.6. *Let us take a 1-safe FLPN* $\mathcal{N}$*, a finite observation* σ *and two constrained unfoldings constructed in two different ways, i.e.* $\mathcal{E}(\mathcal{N}, \sigma)$ *and* $\mathcal{E}_{FL}(\mathcal{N}, \sigma)$*. Both constructions give the same result, i.e.*

$$\mathcal{E}(\mathcal{N}, \sigma) = \mathcal{E}_{FL}(\mathcal{N}, \sigma)\,.$$

Proof. Note that the condition used to construct $\mathcal{E}_{FL}(\mathcal{N}, \sigma)$ is a simplified version of the condition used in the construction of $\mathcal{E}(\mathcal{N}, \sigma)$. Thus, we can simply compare the two conditions and show that they are equal. Consequently, the lemma is true iff the following equivalence is satisfied:

$$\forall e \in E\,,\, \zeta\,(e) \leq \varpi\,(\sigma) \iff \widehat{\zeta}\,(e) \leq \varpi\,(\sigma)\,[\lambda\,(l\,(e))]\,,$$

where E is a set of events of $\mathcal{E}$. Following the last statement, we have two cases to prove.

1. $\forall e \in E\,,\, \zeta\,(e) \leq \varpi\,(\sigma) \implies \widehat{\zeta}\,(e) \leq \varpi\,(\sigma)\,[\lambda\,(l\,(e))]$. This implication is a direct result of the definition of the relation $\leq$ for vectors, and the fact that $\zeta\,(e)$ takes into consideration all events considered by $\widehat{\zeta}\,(e)$.

2. $\forall e \in E\,,\, \widehat{\zeta}\,(e) \leq \varpi\,(\sigma)\,[\lambda\,(l\,(e))] \implies \zeta\,(e) \leq \varpi\,(\sigma)$. In order to prove the implication, we can use the following two properties:

   - $\forall \alpha \in \Sigma \setminus \{\lambda\,[l\,(e)]\}\,,\, \zeta\,(e)\,[\alpha] = \max_{f \in {}^{\bullet\bullet}e}\{\zeta\,(f)\,[\alpha]\}$, and
   - $\zeta\,(e)\,[\lambda\,[l\,(e)]] = \max_{f \in {}^{\bullet\bullet}e}\{\zeta\,(f)\,[\alpha]\} + \chi_{\lambda[l(e)]}$.

   These two properties are a corollary of Lemma 4.5. In both cases we can observe that to compute $\zeta\,(e)$, it is sufficient to consider $\max_{f \in {}^{\bullet\bullet}e}\{\zeta\,(f)\,[\alpha]\}$.

   Moreover, every time we add a new event e, we assume that all events in ${}^{\bullet\bullet}e$ satisfy the observation σ. This implies:

   $$\forall \alpha \in \Sigma \setminus \{\lambda\,[l\,(e)]\}\,,\, \zeta\,(e)\,[\alpha] \leq \varpi\,(\sigma)\,.$$

   Thus, the only component of the observation which can be exceeded when we add e is the one assigned to the events labeled by $\lambda\,[l\,(e)]$.

□

Once we know how to construct a supervisor of FLPN, we have to prove its key properties. We start with Lemma 4.7, and then with Theorem 7. It is worth mentioning that the proof of the theorem also constitutes a frame for similar proofs, *e.g.* for a generic 1-safe Petri net which is presented later in the work.

Lemma 4.7. *Let $\mathcal{N}$ be a 1-safe FLPN with silent transitions and $\mathcal{E}\,(\mathcal{N}, \sigma) = \langle B, E, F, l\rangle$ a constrained unfolding of $\mathcal{N}$ guided by an observation σ. Moreover, let us consider a sequence of configurations $\mathbf{C} = (C_0, \ldots, C_n)$, such that:*

- $C_n \subseteq E$,
- $C_0 = past\,(e)$, *where* $e = \epsilon$-companion (e'),
  $C_1 = past\,(e')$, *where* e' *is an* ϵ-terminal *event,*
  $C_i = C_{i-1} \oplus I\,[past\,(e') \setminus past\,(e)]$.

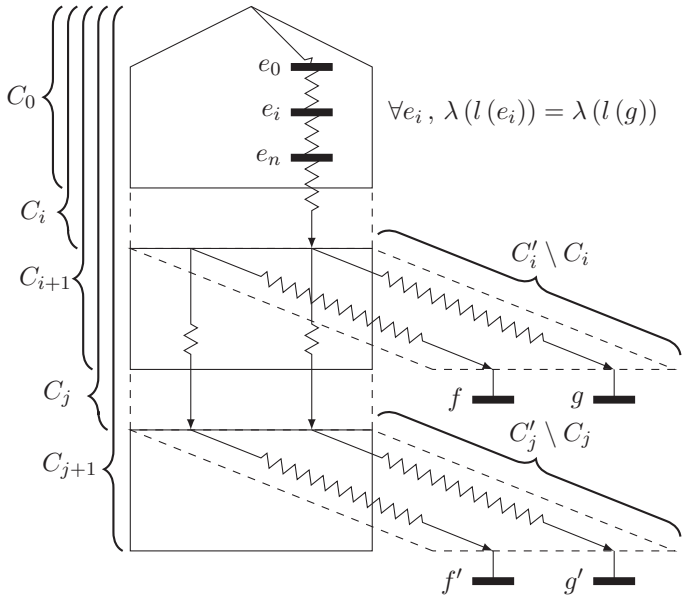


Figure 4.7: Illustration of the proof of Lemma 4.7.

Then the following statement is true:

$$\forall C_i, C_j \in \mathbf{C}\,,\; futures_{|E}\left(C_i\right) = I\left[futures_{|E}\left(C_j\right)\right].$$

Proof. Note that the statement of the lemma can be written as follows:

$$\forall C_i, C_j \in \mathbf{C}\,,\; \forall F \subseteq futures_{|E}\left(C_j\right), C_i \oplus I\left(F\right) \subseteq E.$$

We use that fact and we split the proof into two parts showing two possible cases.

1. $\forall C_i, C_j \in \mathbf{C}, i < j \implies \forall F \subseteq futures_{|E}\left(C_j\right), C_i \oplus I\left(F\right) \subseteq E$

   From the assumptions of the lemma, we know that, if $i < j$, then $C_i \subset C_j$. We also know that both configurations C_i and C_j are histories of some events, and $l\left(\mathrm{cut}\left(C_i\right)\right) = l\left(\mathrm{cut}\left(C_j\right)\right)$.

   From Lemma 4.2, we can observe that each condition of $\mathrm{cut}\left(C_j\right)$ is causally preceded by a corresponding condition from $\mathrm{cut}\left(C_i\right)$, *i.e.* by the condition associated with the same place in the net. This induces the fact that the conditions of $\mathrm{cut}\left(C_i\right)$ have Parikh vectors which are not greater than the respective conditions of $\mathrm{cut}\left(C_j\right)$.

   Let us take any configuration $C'_i \subseteq E$, such that $C_i \subset C'_i$. Then, let us try to extend it with an event $f \in E$, *i.e.* the result should be $C'_i \oplus \{f\}$. Then, consider an analogically constructed extensions, that is to say $C'_j = C_j \oplus I\left(C'_i \setminus C_i\right)$ and $C'_j \oplus \{f'\}$, where $f' \equiv I\left(\{f\}\right)$. Now,

consider the maximal subset B_i of $cut\,(C_i)$, such that all conditions of B_i are causal predecessors of f. Analogically, we construct a subset $B_j \subseteq cut\,(C_j)$ for the event f'. Since $C'_i \setminus C_i$ and $C'_j \setminus C_j$ are isomorphic, we can note that $l\,(B_i)$ and $l\,(B_j)$ are identical.

Given all the properties above, we can observe that the Parikh vector of f is equal or less than the one of the event f'.

2. $\forall C_i, C_j \in \mathbf{C}, i < j \implies \forall F \subseteq futures_{|E}\,(C_i)\,, C_j \oplus I\,(F) \subseteq E$

Let us consider once again the situation described in the first part of the proof. Thus, there are the two configurations C'_i and C'_j. This time, we try to add an event g which extends C'_i to $C''_i = C'_i \oplus \{g\}$, and an event $g' = I\,(g)$ which produces $C''_j = C'_j \oplus \{g'\}$. We will conduct the proof by contradiction, *i.e.* we assume that the event g can extend C'_i whereas the event g' cannot extend C'_j. This would mean that there are more events labelled by $\lambda\,[l\,(g)]$ before g' than before g. Recall that Definition 51 shows that, every time we add a new event, it is sufficient to check only one component of its Parikh vector, which is assigned to the action of the new event. Moreover, from Lemma 4.5, we know that, every time we add a new event h to a process of *FLPN*, all transitions in the process with the label $\lambda\,[l\,(h)]$ causally precede h. Consequently, we are sure that, before g, there are no events labeled by $\lambda\,[l\,(g)]$ which are parallel to g. Otherwise, the events would be in conflict with g. Thus, all the events in C'_i labeled by $\lambda\,[l\,(g)]$ are before g. Since $C_j \setminus C_i$ only consists of unobservable events, we can observe that all events labeled by $\lambda\,[l\,(g)]$ in C_i are before g'. Also note that $C'_i \setminus C_i$ is isomorphic to $C'_j \setminus C_j$. As a result, the number of events in C'_j labeled by $\lambda\,[l\,(g)]$ before g' cannot be greater than the number of events in C'_i labeled by $\lambda\,[l\,(g)]$ before g.

□

Intuitively, Lemma 4.7 shows that, if we consider configurations resulting from a number of repetitions of an unobservable loop, the possible future for each of them is always the same. As a direct result of Lemma 4.7, we get the property showing that, in the construction process of constrained prefix of *FLPN,* it is sufficient to repeat each unobservable loop only once. This property is presented in the following Theorem 7.

THM **Theorem 7.** Let $\mathcal{N} = \langle P, T, W, m_0 \rangle$ be a 1-safe free-labeled Petri net with silent transitions. Consider $\mathcal{S}\,(\mathcal{N}, \sigma)$ as a supervisor (from Definition 50) of the net $\mathcal{N}$ for the finite observation σ. The unfolding of $\mathcal{S}\,(\mathcal{N}, \sigma)$ is denoted by $\mathcal{U}_{\mathcal{S}}\,(\mathcal{N}, \sigma) \equiv \mathcal{U}\,(\mathcal{S}\,(\mathcal{N}, \sigma))$. The following is always satisfied:

$$processes\,[\mathcal{E}\,(\mathcal{N}, \sigma)] =_1 processes\,[\mathcal{U}_{\mathcal{S}}\,(\mathcal{N}, \sigma)]\,.$$

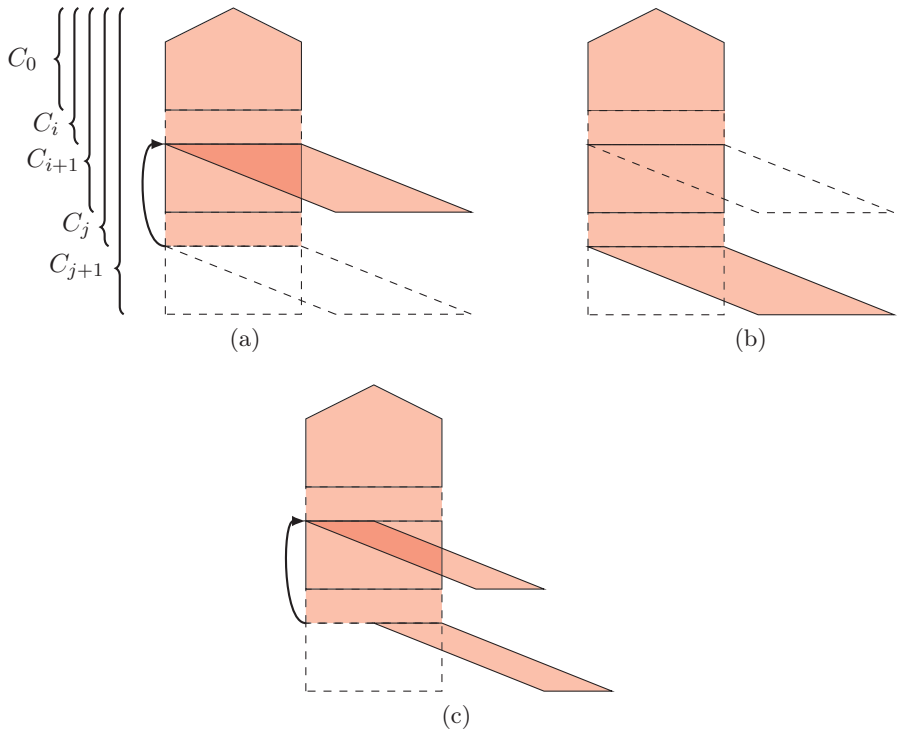


Figure 4.8: Illustration of the proof of Theorem 7.

The relation $=_1$ means that both arguments are isomorphic up to names of conditions and events.

Proof. We can split the proof into two parts: in the first case, we show that all processes of $\mathcal{E}(\mathcal{N}, \sigma)$ are coded in $\mathcal{U}_{\mathcal{S}}(\mathcal{N}, \sigma)$. Then, we prove the inverse *i.e.* all processes of $\mathcal{U}_{\mathcal{S}}(\mathcal{N}, \sigma)$ are coded in $\mathcal{E}(\mathcal{N}, \sigma)$. For the sake of simplicity, we skip arguments in the structures.

1. $\forall \pi_1 \in processes[\mathcal{E}(\mathcal{N}, \sigma)], \exists \pi_2 \in processes[\mathcal{U}_{\mathcal{S}}(\mathcal{N}, \sigma)], \pi_1 =_1 \pi_2$

   First, let us briefly recall the chain of operations which leads from $\mathcal{E}$ to $\mathcal{U}_{\mathcal{S}}$. From Definition 49, we know that $\mathcal{P}$ is a prefix of $\mathcal{E}$. In turn, $\mathcal{P}$ is a base for construction of $\mathcal{S}$, and thus the unfolding $\mathcal{U}_{\mathcal{S}}$. We can observe in Definition 50 that, by starting from the initial event up to ϵ-terminal events and the cuts associated with the events, both structures $\mathcal{E}$ and $\mathcal{U}_{\mathcal{S}}$ are isomorphic. The problem arises with an ϵ-terminal event in $\mathcal{P}$. From Definition 50, we know that ϵ-terminal events imply cuts which are subjects to the operation of fusion (see section 4.2.4). Moreover, we know that such an ϵ-terminal event has its ϵ-companion event which generates a companion cut. Following that, we can take an ϵ-terminal event e', a cut $\mathrm{cut}(past(e'))$ and its companion cut

cut $(past\,(e))$, where $e = \epsilon$-companion (e'). Now, we can use the fact that there was a fusion of conditions, and we will continue the process from the companion cut. Then, from Lemma 4.7, we know that both cuts have exactly the same possible futures in $\mathcal{E}$. As we mentioned, if there were some events E executed after the cut, they can also be executed after the companion cut, resulting in $past\,(e) \oplus I\,(E)$. Note that, except e, there may be other ϵ-terminal events in E. However, this fact does not disrupt the presented procedure as we can consider a subset E' of E without any ϵ-terminal events. Then, we can continue adding the rest of the events of $E \setminus E'$ to $past\,(e) \oplus I\,(E')$. Every time there is a new ϵ-terminal event, we simply repeat the whole procedure. Thus any process which is stored by $\mathcal{E}$ is present in $\mathcal{U}_{\mathcal{S}}$. This case is briefly illustrated in 4.8a.

2. $\forall \pi_1 \in processes\,[\mathcal{U}_{\mathcal{S}}\,(\mathcal{N}, \sigma)]\,, \exists \pi_2 \in processes\,[\mathcal{E}\,(\mathcal{N}, \sigma)]\,, \pi_1 =_1 \pi_2$

   We have shown that we can recreate all processes of $\mathcal{E}$ in $\mathcal{S}$. Now, we have to take all the other processes of $\mathcal{S}$ which we did not analyze in the previous case and show that they are stored by $\mathcal{E}$. Let us recall a cut associated with the ϵ-terminal event e'. We showed that the future of the cut and its companion cut are the same. We also demonstrated that we "moved" with the continuation of the process to the companion cut every time there was such a cut. We present three possible cases which were not analyzed by us in the previous situation and which are sufficient to prove the second case of the proof. In the following part, we assume that $\pi_1 = \langle B_1, E_1, F_1, l_1 \rangle$, $\pi_2 = \langle B_2, E_2, F_2, l_2 \rangle$.

   (a) If there is an ϵ-terminal event, we continue the process as long as it is possible, *i.e.* until there is a successor of the ϵ-terminal event which could not be added to $\mathcal{P}$ due to its construction rules, and thus which is not stored by $\mathcal{S}$. In this case, we know that the created process is present in $\mathcal{E}$ as it is a prefix of $\mathcal{P}$.

   (b) The second case is when there is a cut in a process π of $\mathcal{S}\,(\mathcal{N}, \sigma)$, and when one decides to continue the process using a mix of transitions that are after the cut and transitions that are after the companion cut. We can notice that the same result can be obtained if the process π has its continuation starting from the companion cut.

   (c) We can imagine a process in which, after an ϵ-terminal event, some events from E_1 were added after the cut of the event, and some events from E_2 were added after the companion cut. As in the previous cases, here we use the fact that the conditions of the two cuts are merged. Now, if we consider the new extension, we can note that it is impossible to add a new event which would

use preconditions from E_1 and E_2 at the same time. This would generate a conflict.

Figures 4.8b and 4.8c respectively depict the cases a), b) and c).

□

Given the above theorem, we can prove the two following properties.

Lemma 4.8. ***(Correctness)*** *Let $\mathcal{N}$ be a 1-safe free-labeled Petri net and σ a finite observation. Let $\mathcal{S}(\mathcal{N}, \sigma)$ be a supervisor. Any event e of $unfold[\mathcal{S}(\mathcal{N}, \sigma)]$ satisfies the observation σ.*

Proof. The proof is a direct consequence of Theorem 7. □

Lemma 4.9. ***(Completeness)*** *Let $\mathcal{N}$ be a 1-safe free-labeled Petri net and σ a finite observation. Let $\mathcal{S}(\mathcal{N}, \sigma)$ be a supervisor. For any observation σ, $\mathcal{S}(\mathcal{N}, \sigma)$ represents all possible processes which satisfy the observation.*

Proof. The proof is a direct consequence of Theorem 7. □

Example 26. Let us consider the free-labeled Petri net $\mathcal{N}$ in Figure 4.1a and an observation $\sigma = \{t_4, t_4\}$ (we use names of transitions as labels). The observable events and transitions are drawn with a bold line. Figure 4.9 represents a small fragment of the constrained unfolding $\mathcal{E}(\mathcal{N}, \sigma)$ which is infinite in general. In Figure 4.10, there is a constrained prefix $\mathcal{P}(\mathcal{N}, \sigma)$. The dashed arrows indicate the places which are merged in the related supervisor $\mathcal{S}(\mathcal{N}, \sigma)$. The symbol $\bullet$ is used to indicate the ϵ-terminal events, $\bigstar$ is used to distinguish conditions in their cuts which are merged. Events and conditions inside foldable structures are filled in gray. After the fusion of appropriate conditions (accordingly to Definition 50), we obtain the supervisor presented in Figure 4.11. EXM

4.4.2 Removal of incomplete explanations

As we mentioned before, a constrained unfolding (or prefix) may contain explanations which are not complete. We propose a procedure 4.3 which can be used to remove such explanations. The basic idea of the algorithm is to remove all events which do not belong to any complete explanation of σ. All the events for which we are sure that they belong to some complete explanations are stored in the set *Accepted*. In the first step (item 1.), we put in *Accepted* events such that their predecessors completely satisfy σ. Then, we analyze the rest of the events step by step considering all maximal events which are not in *Accepted* (items 2.-4.). At each step, we remove unnecessary maximal events, *i.e.* which do not belong to any complete explanation. We repeat the whole procedure until all the remaining events are in *Accepted*.

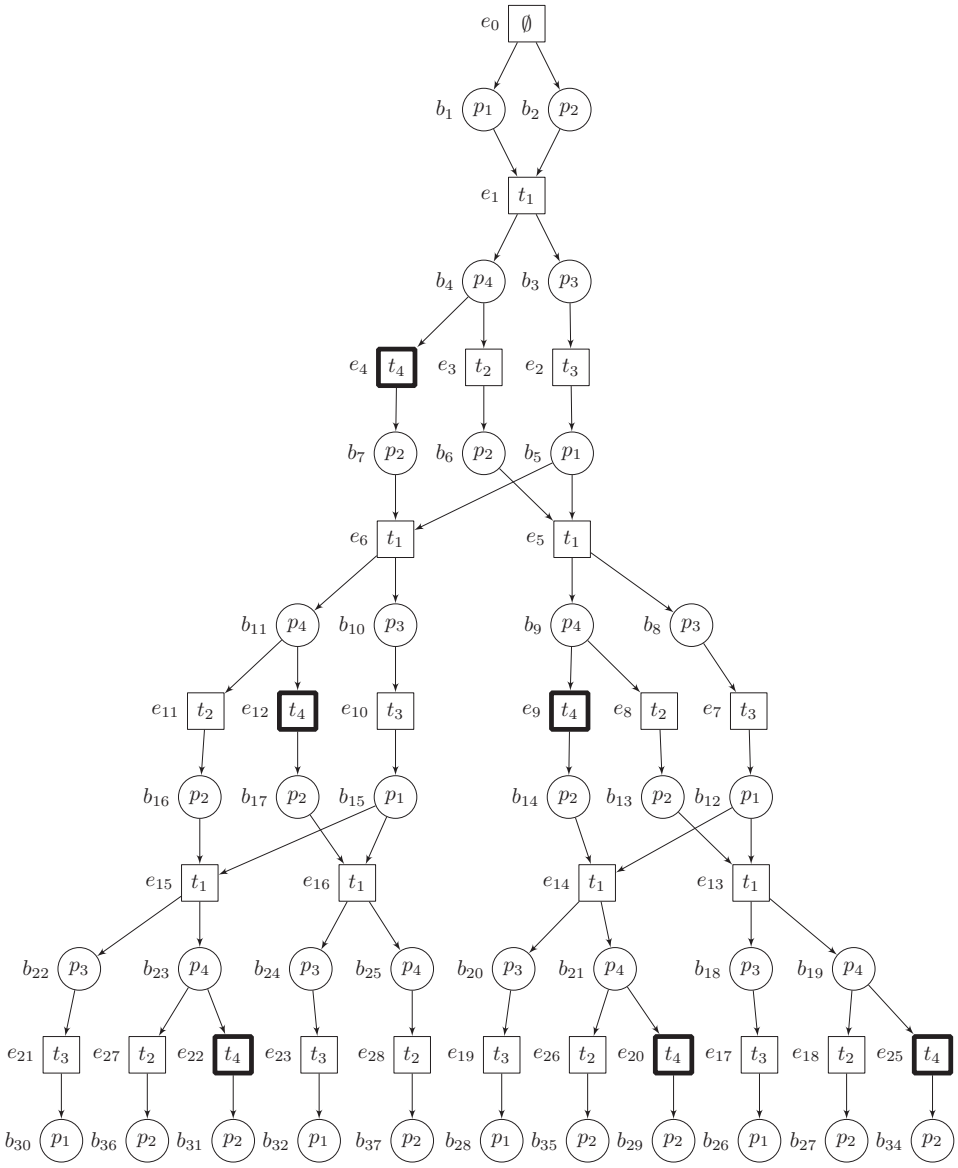


Figure 4.9: Prefix of the constrained unfolding of the Petri net in Figure 4.1.

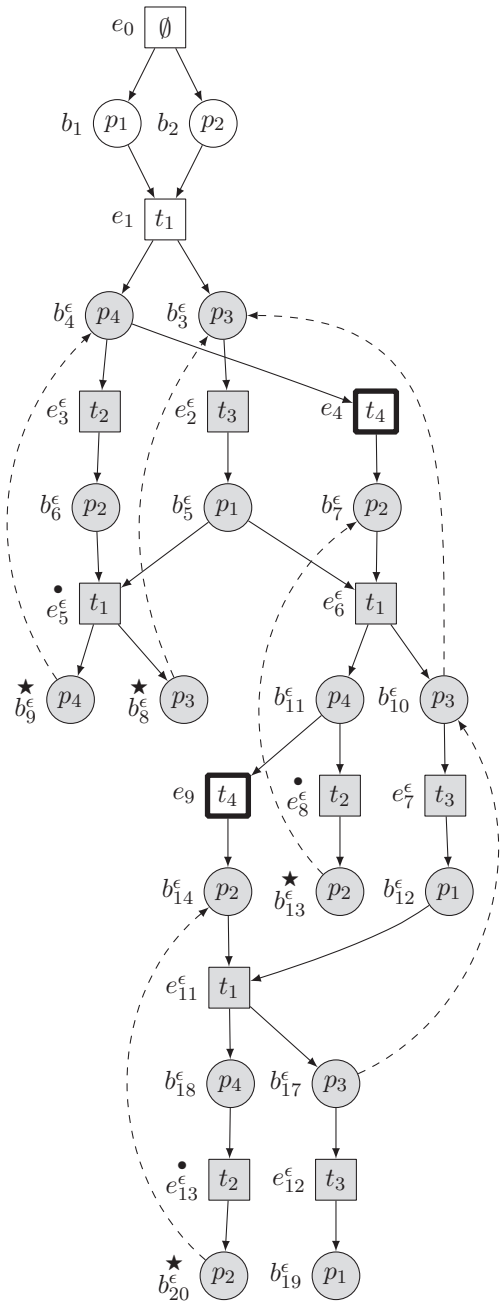


Figure 4.10: Constrained prefix of the Petri net in Figure 4.1.

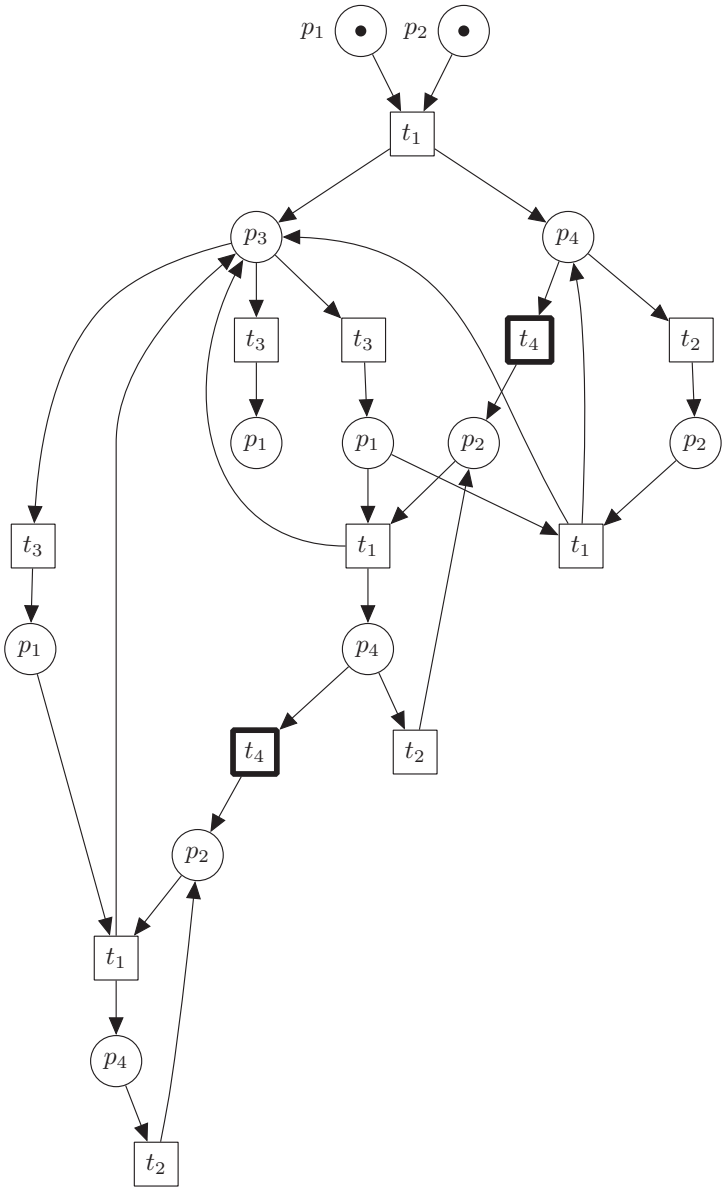


Figure 4.11: A supervisor based on the constrained prefix in Figure 4.10.

Algorithme 4.3 Removal of incomplete explanations from constrained unfolding.

Input: A constrained prefix $\mathcal{P}(\mathcal{N}, \sigma)$
Output: A constrained prefix $\mathcal{P}(\mathcal{N}, \sigma)$ with removed incomplete explanations

1. Take all the events that **completely** satisfy σ and put them in *Accepted*.
2. Also add to *Accepted* the events that precede events in *Accepted*.
3. Take all the maximal events of $\mathcal{P}(\mathcal{N}, \sigma)$ such that they are **not** in *Accepted* and put them in F.
4. For each $e \in F$ do
   (a) Let $\mathcal{G}$ be a set of all co -sets which contain e
   (b) For each $C \in \mathcal{G}$ do
      - If $\bigcup_{f \in C} past(f)$ completely satisfies the observation then
        $Accepted = Accepted \cup (C \cap F)$
        $F = F \setminus C$
        goto **2**
   (c) Remove e from $\mathcal{P}(\mathcal{N}, \sigma)$
5. If all events which rest are in *Accepted*, then **stop**. Otherwise goto **2**

Algorithme 4.4 Elimination of redundant places and transitions from a supervisor.

Input: A supervisor $\mathcal{S}(\mathcal{N}, \sigma) = \langle P, T, W, l \rangle$
Output: A supervisor with removed unnecessary redundant transitions and places, denoted by $\mathcal{S}_{reduced}(\mathcal{N}, \sigma)$

1. Take a set $\mathcal{T} \subseteq 2^T$ such that $\forall T_i \in \mathcal{T}, \forall t_j, t_k \in T_i, l(t_j) = l(t_k) \wedge {}^\bullet t_j = {}^\bullet t_k$
2. If $\mathcal{T} = \emptyset$ then **stop**
3. For each $T_i \in \mathcal{T}$ do
   (a) Merge all transitions in T_i and their postsets with respect to the names of the original places in $\mathcal{N}$.
4. Goto 1

4.4.3 Elimination of duplicate processes

As we could note in Example 26, sometimes some unnecessary duplications of transitions may appear in supervisors. For example, in Figure 4.11, we can observe three transitions going out of the same place and associated with the same transition t_3 of the underlying Petri net in Figure 4.1a. We propose a method to remove transitions and places which produce duplicates of some explanations.

Lemma 4.10. *The algorithm 4.4 terminates.*

Proof. The input $\mathcal{S}(\mathcal{N}, \sigma)$ is finite. At any time, we can only remove transitions and places; we do not add new places or transitions. Thus, at some point of the algorithm execution, the condition from line 2. will be satisfied. □

Lemma 4.11. *The set of processes of the supervisor is preserved.*

Proof. In the algorithm 4.4, no unique transition is removed. Only transitions of the supervisor which are associated with the same transition of the underlying Petri net are merged together with their postsets which also have to be identical (line 1. of the algorithm). □

Lemma 4.12. *There are no redundant processes in the final supervisor of the algorithm 4.4 with respect to the underlying net $\mathcal{N}$.*

In other words, if there are two isomorphic processes up to the naming of transitions and places in $\mathcal{N}$, it means that they are created using the same transitions of the supervisor $\mathcal{S}_{reduced}(\mathcal{N}, \sigma)$.

Proof. We can note that the situation contradicting the lemma is impossible since we removed all duplications of the transitions from $\mathcal{S}(\mathcal{N}, \sigma)$. □

The time complexity of the algorithm is linear in the size of the supervisor.

Example 27. In Figure 4.12, we can find a reduced version of the supervisor presented in Figure 4.11. As we can observe, the new supervisor is smaller than the previous one. EXM

4.5 Constrained unfolding of 1-safe Petri nets

4.5.1 Construction of constrained unfolding

Unfortunately, the solution adopted in the previous chapter does not work in the general case of 1-safe Petri nets with silent transitions. To understand why, let us look closer at the net presented in Figure 4.13. The net is a 1-safe Petri net. It consists of a number of equal segments. However, we can note that all its transitions have the same label "a", therefore it is not a free-labeled Petri net. However, we note the fact that all visible transitions which have the same label does not affect the generality of the solution that we present below.

Let us consider the constrained prefix in Figure 4.14 constructed on the base of the net in Figure 4.13. The prefix is based on the observation $\sigma = \{a, a, a, a\}$. Note that if we would apply the construction procedure of the constrained prefix used for 1-safe free-labeled Petri nets, the event $e_{2,4}$ would be an ϵ-terminal event. Moreover, the resulting supervisor would look like the one in Figure 4.15.

Now, let us consider the unfolding of such a supervisor. It is very similar to the one shown in Figure 4.14, except for one important detail. Note that we have one invisible loop in the underlying Petri net. After each repetition of the loop, according to the results presented in Figure 4.15, it is possible to run two transitions: t_3 and t_4. However, when we look at Figure 4.14, we can observe that the supervisor does not meet the conditions imposed by the observation σ.

To understand the problem which appears in the example above, we need to look at the pre-condition of transitions t_3. Note that with each repetition of the unobservable loop, the number of observable events preceding the event associated to the transition t_3 is being changed. Thus, before the first execution of the unobservable loops, the number of observable events is equal to 0; after one loop, there is one observable event before t_3; after two loops, there are two observable events, etc. In other words, after each repetition of the loop, the number of observable events before the condition corresponding

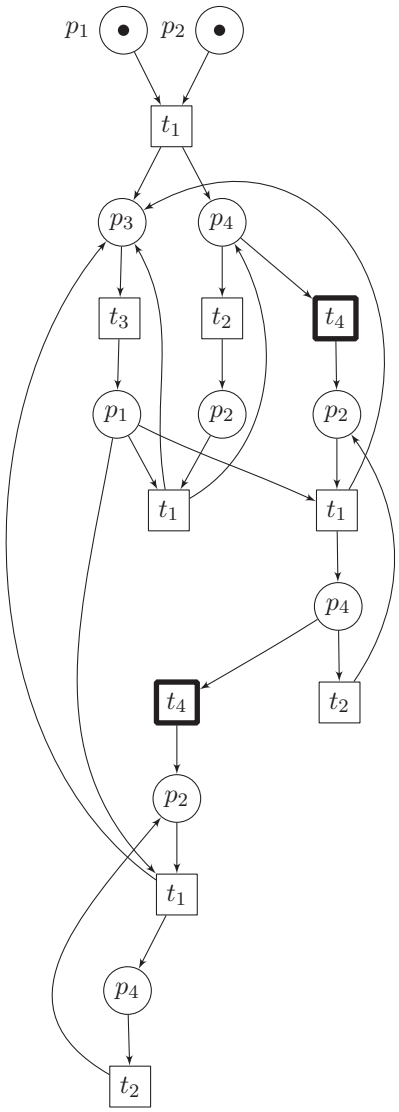


Figure 4.12: A reduced supervisor of the Petri net in Figure 4.1.

to place p_1 is increased by one. After four and more repetitions, the number of events is four.

As a result, the value of the Parikh vector for an event associated with t_3, which potentially can occur after each repetition of the loop, is also changed. This, in turn, directly affects the set of events which occur after the event. It should be noted that, depending on the number of loop iterations in our example, the number of events following t_3 may differ. This way, after a single or double execution of the loop, we can execute t_3 and t_4; after a triple loop execution, only t_3 can be executed; and finally, after four or more repetitions of the loop, we can no longer fire the transition t_3. This example is precisely illustrated in Figure 4.14.

After the analysis of the above example, two questions may arise: in general, what is the necessary number of repetitions of the ϵ-loop so that we can be confident that the state reached after the last repetition is stable? Through a stable condition after the loop, we mean a state whose future is the same as the future of the state before the loop. Of course, we have to take into account an observation. Is the number of iterations always finite?

To answer both questions, for the most general case of 1-safe Petri nets, let us point out that the key element to achieve the stability condition for the ϵ-loops is the Parikh vector for events which may occur immediately after a certain number of repetitions of the ϵ-loop. Namely, let s_1 be a state reached after m iterations of the loop, and s_2 a state reached after m' iterations of the loop. We can say that both states s_1 and s_2 can form a stability condition necessary to create a supervisor if the following situations are fulfilled: 1) for every transition available from the state s_1, there is an identical transition available from s_2, and 2) for each such a pair of transitions, their associated events have the same sets of observable events in their past. This way, we can inductively show that, under certain circumstances, a set of possible configurations is the same after each repetition of the unobservable loop in the supervisor.

Above all, if we take the state before the first performance of the ϵ-loop and the state after n iterations of the loop, we can observe that the number of observable events is actually not increased. The only observable events occur before the first repetition of the loop. With successive repetitions, there are new connections between the events and conditions of the initial state after each repetition of the loop. The algorithms we propose identify such possible connections between conditions of the initial and final states (concerning executions of the loop).

Below, in the following two sections, we present two possible solutions to the problem.

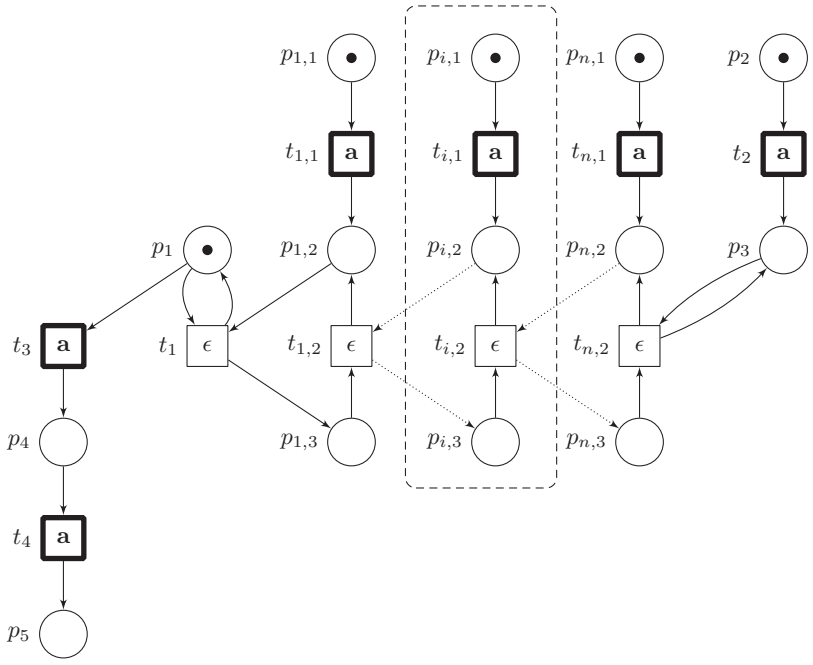


Figure 4.13: A schema of 1-safe Petri net.

4.5.2 Approach 1

In this approach, we will show how we can identify a number of repetitions of a silent loop before each repetition gives a state with the same futures. In the first approach, we propose a solution with a fixed number of repetitions. For this reason, we introduce a new definition of an ϵ-terminal event.

$\pi_{|X}$ denotes a process consisting of events in X.

DEF **Definition 52.** **(ϵ-terminal event of type 3)** Let $\beta = \langle B, E, F, l\rangle$ be a branching process of a 1-safe Petri net $\mathcal{N} = \langle P, T, W, m_0\rangle$. Let us consider a sequence of configurations $\vec{C} = (C_0, \ldots, C_n)$ such that:

- $\forall C_i, C_{i+1}\,,\ C_i \subseteq C_{i+1}$,
- $C_n \subseteq E$,
- $C_0 = past\,(e_0)$, where $e_0 = \epsilon$-companion (e_1),
  $C_1 = past\,(e_1)$, where e_1 is an ϵ-terminal event from Definition 48,
  $C_i = C_{i-1} \oplus I\,(past\,(e_1) \setminus past\,(e_0)) = past\,(e_i)$, for $i \in [2, n]$,
- $n = |\mathrm{cut}\,(C_0)| - 1$.

We call e_n an ϵ-terminal event of type 3. In other words, ϵ-terminal$^3\,(e_n)$ is true if e_n is an ϵ-terminal event of type 3. Analogically as for ϵ-terminal events, we define an ϵ-companion event, *i.e.* ϵ-companion$^3\,(e_n) = e_0$. To

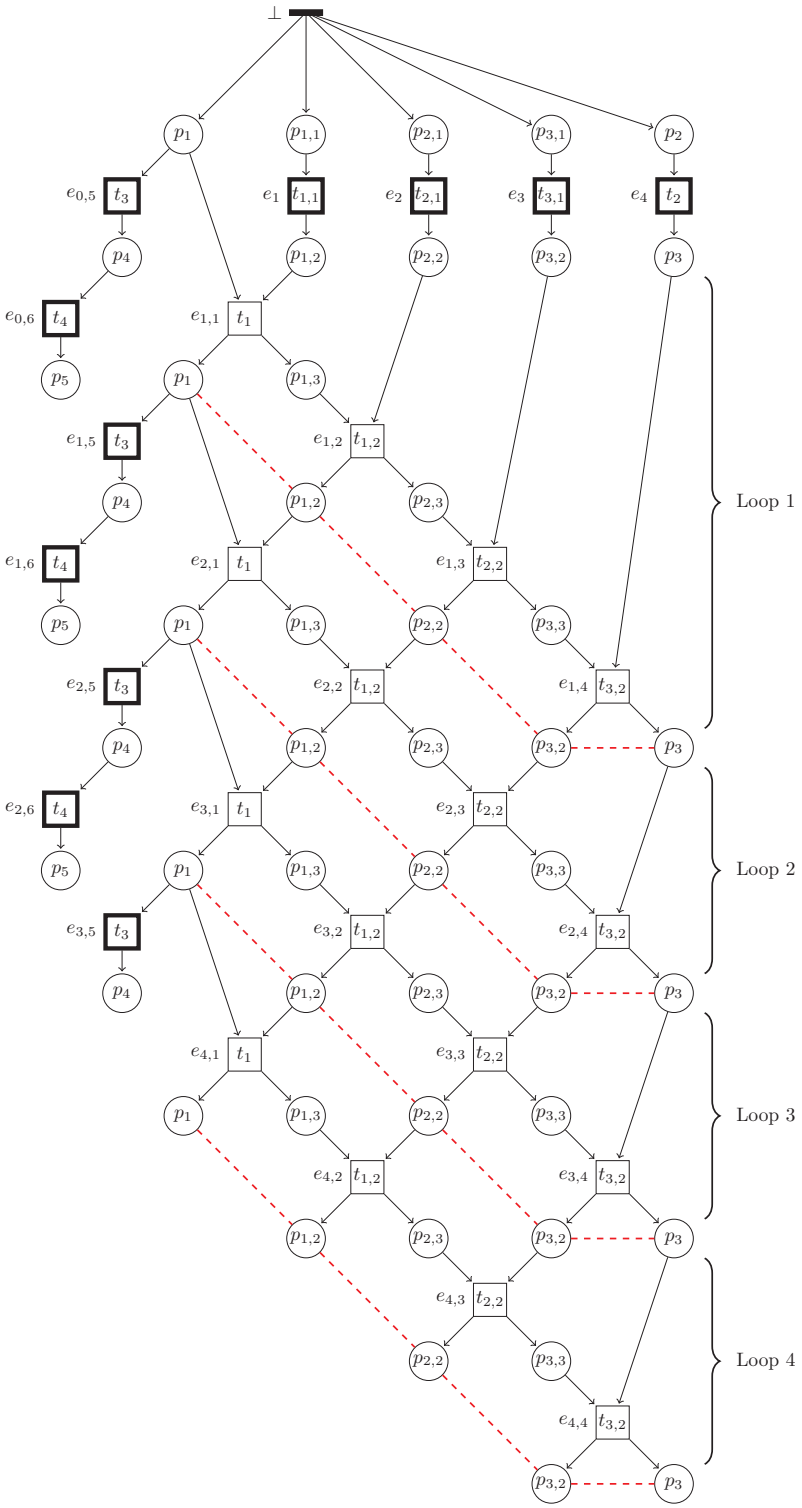


Figure 4.14: A prefix of the constrained unfolding of the Petri net in Figure 4.13.

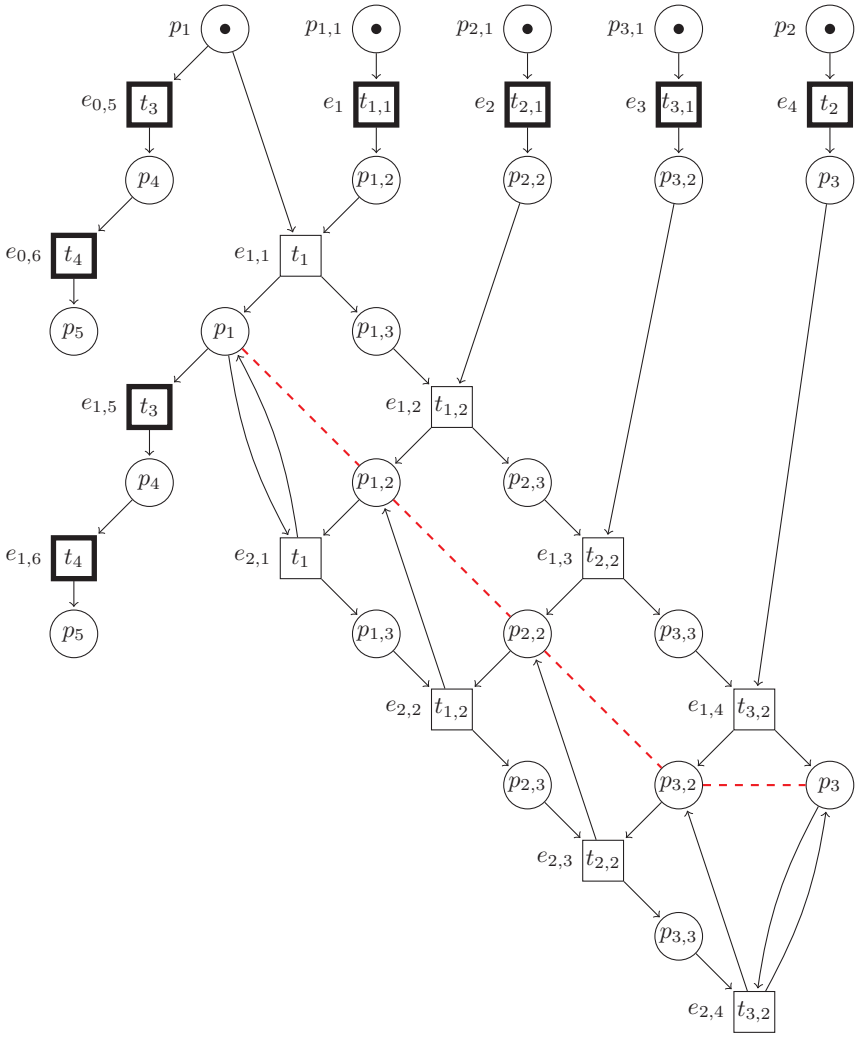


Figure 4.15: Wrong supervisor of the net in Figure 4.13.

simplify the notation, we also define the secondary ϵ-companion, *i.e.* ϵ-companion$_2^3(e_n)$ to denote e_1. ♦

The idea behind the definition is quite simple. Intuitively, when we look into the solution proposed for free-labeled Petri nets, we can observe that we stop extending an ϵ-terminal event which is produced by one repetition of the associated unobservable loop. As we already showed before, in the case of general 1-safe Petri nets, the construction is incorrect. In the solution presented in this section, each silent loop has to be repeated $n-1$ times, where n is the number of conditions reached after each repetition of the loop. In other words, it is the number of tokens in the associated marking. As we will see in the following lemma, this is sufficient to guarantee that the construction is correct and correctly stores all explanations of the related constrained unfolding.

Lemma 4.13. *Let $\mathcal{N}$ be a 1-safe Petri net and $\mathcal{E}(\mathcal{N},\sigma) = \langle B, E, F, l\rangle$ a constrained unfolding of $\mathcal{N}$ guided by an observation σ. Let us consider a sequence of configurations $\vec{C} = (C_0, \ldots, C_n)$ such that:*

- $\forall C_i, C_{i+1}\,,\ C_i \subseteq C_{i+1}$,
- $C_n \subseteq E$,
- $C_0 = past\left(e^0\right)$, *where* e^0 *is an* ϵ-terminal *event of type 3,*
  $C_i = C_{i-1} \oplus I\left(past\left(f\right) \setminus past\left(e^0\right)\right)$, *where* $f = \epsilon$-companion$_2^3\left(e^0\right)$,

Then the following statement is true:

$$\forall C_i, C_j \in \vec{C}\,,\ futures_{|E}\left(C_i\right) = futures_{|E}\left(C_j\right).$$

Proof. Each configuration C_i in $\vec{C}$ represents an extension of C_0 after ith iteration of an unobservable loop. By S_i, we denote cut$\left(C_i\right)$. Remark that, for all configurations in $\vec{C}$, $l\left(S_i\right)$ is the same and $|S_i| = m$. Now let us consider a condition $b^0 \in S_0$ and all its counterparts in the subsequent sets S_i, denoted by b^i. By the counterparts, we understand conditions associated with the same place (according to $\mathcal{N}$). Moreover, we will consider a subset $P_l^k \subseteq S_k$, such that P_l^k is a maximal set of predecessors of b^l. Note that $P_l^k = P_{l+o}^{k+o}$.

Knowing the properties presented above, we can analyze how the set of observable events changes for subsequent conditions in S_i. First, note that P_l^0 can only grow in number of elements as l is being increased. This is due to Lemma 4.2 from which we know that b^i causally precedes b^j, when $i < j$. As P_l^0 grows, note that the number of possible observable events of b_l grows. However, we know that such a growth is bounded by m. Each time the loop is repeated, there are two possible situations:

1. $P_l^0 = P_{l+1}^0$, this means that the set of observable events before b^l and b^{l+1} is the same. Thus, we can stop repeating the loop,

2. $P_l^0 \subset P_{l+1}^0$, in this case the number of observable events before b^l and b^{l+1} is different and we have to continue repeating the loop to reach a stable set of observable events for b^i.

The more interesting for us is the second situation in which we have to repeat the loop. Let us check what is the maximal necessary number of repetitions. Note that the loop has to be repeated iff we have the second situation after each iteration. In order to obtain the number which we look for, we can note that, after each repetition, $\left|P_l^0\right|$ grows at least by one, *i.e.* $\left|P_l^0\right| + 1 = \left|P_{l+1}^0\right|$, where $\left|P_0^0\right| = 1$. This way, we can deduce that, after $m-1$ iterations, $\left|P_m^0\right| = m$, which is the boundary we mention before. To conclude, for $p > q \geq m-1$, $\left|P_p^0\right| = \left|P_q^0\right|$. □

Having the above property, we can define a constrained prefix for 1-safe Petri nets.

DEF **Definition 53. (Constrained prefix)** Let $\mathcal{N}$ be a 1-safe Petri net, and let σ be a finite observation. A constrained unfolding under partial observation $\mathcal{E}_{PO}^3(\mathcal{N}, \sigma)$ is a maximal branching process such that:

$$\forall e \in \mathcal{E}_{PO}^3(\mathcal{N}, \sigma) \begin{cases} \nexists f \in E,\ \epsilon\text{-terminal}^3(f) \wedge e > f \\ \zeta(e) \leq \varpi(e) \end{cases}$$

♦

4.5.3 Approach 2

In section 4.5.2, we introduced a solution in which we had to repeat each silent loop a certain *fixed* number of times before we could stop extending a given ϵ-terminal event. Below, we will describe how the previously presented approach can be improved by minimizing the number of repetitions of the unobservable loops.

The algorithm that we propose uses a single segment of the ϵ-loop. In Figure 4.18, we can see such a segment from the example in Figure 4.14. The idea is to transform it into a weighted graph with which we can compute the number of iterations of the loop required to achieve all the connections between the respective conditions. The outline of the algorithm is in 4.5.

Having the function $reps(G)$, we can define a new modified version of ϵ-terminal event.

DEF **Definition 54.** (ϵ-terminal **event of type 4)** Let $\beta = \langle B, E, F, l \rangle$ be a branching process of a 1-safe Petri net $\mathcal{N} = \langle P, T, W, m_0 \rangle$. Let us consider a sequence of processes $\vec{C} = (C_0, \ldots, C_n)$ such that:

Algorithme 4.5 The computation of the number of ϵ-loop iterations.

Input: A segment G of an ϵ-loop

Output: A number of repetitions of the loop, denoted by $reps\,(G)$, necessary to correctly terminate the constrained prefix

1. Convert the ϵ-loop into a directed graph and then calculate its transitive closure.
2. We eliminate all the vertices (and edges adjacent to them) except for vertices of the initial and final states of the loop.
3. For all other edges, we assign cost 1.
4. In addition, the conditions attached to the same places in the underlying net have to be connected with edges with cost 0. These edges are used to represent repetitions of the ϵ-loop.
5. Having the directed graph, we compute all the paths of the minimum cost between all pairs (v_1, v_2) such that v_1 belongs to the initial state, and v_2 to the final state. To solve this problem, we can use *e.g.* Floyd-Warshall algorithm (see [36]).
6. From all the existing paths, we choose the one with the highest cost. This cost corresponds to the number of iterations required to stabilize the state.

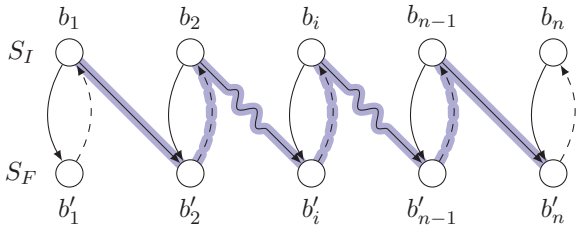


Figure 4.16: Illustration for the proof of Lemma 4.14.

- $\forall C_i, C_{i+1}\,,\ C_i \subseteq C_{i+1}$,
- $C_n \subseteq E$,
- $C_0 = past\,(e_0)$, where $e_0 = \epsilon\text{-companion}\,(e_1)$,
  $C_1 = past\,(e_1)$, where e_1 is an ϵ-terminal event from Definition 48,
  $C_i = C_{i-1} \oplus I\,(past\,(e_1) \setminus past\,(e_0)) = past\,(e_i)$, for $i \in [2, n]$,
- $n = reps\,(past\,(e_1) \setminus past\,(e_0))$.

We call e_n an ϵ-terminal event of type 4. In other words, $\epsilon\text{-terminal}^4\,(e_n)$ is true if e_n is an ϵ-terminal event of type 4. Analogically as for ϵ-terminal events, we define an ϵ-companion event, *i.e.* $\epsilon\text{-companion}^4\,(e_n) = e_0$. To simplify the notation, we also define the secondary ϵ-companion, *i.e.* $\epsilon\text{-companion}^4_2\,(e_n)$ to denote e_1. ♦

Below, we prove that the new algorithm actually always gives a smaller number of iterations of unobservable loops.

Lemma 4.14. Let us consider Definitions 52 and 54. The following is true:

$$reps\,[past\,(e_1) \setminus past\,(e_0)] \leq |\mathrm{cut}\,(C_0)| - 1$$

Proof. Because the result of the *reps* function may differ for different loops, we take the worst case in which the value is the greatest. We prove that the number obtained with the function is not greater than $|\mathrm{cut}\,(C_0)| - 1$.

Let us consider a directed graph G which is constructed on the base of a segment of ϵ-loop, *i.e.* $past\,(e_1) \setminus past\,(e_0)$ in the Lemma. It is the same graph which is used in the procedure for the computation of $reps\,(G)$ (see 4.5). The vertices of the graph can be divided into two groups: the initial vertices S_I and the final vertices S_F. The number of vertices in each group is equal to $|cut\,(C_0)| = m$. There are only edges from S_I to S_F with cost 1. The edges with cost 0 connect vertices assigned to the same places in the underlying Petri net. There is no path from a vertex of S_F to a vertex of S_I without at least one edge of cost 0. There are no edges between vertices inside S_I (the same is true for S_F).

Let us take any two vertices in G such that there exists a path between them. Moreover, we assume that it is the path with the least possible cost which is at the same time the greatest cost in such a graph. When we take into consideration all above properties, we can observe that the cost is equal to $m-1$, which proves the lemma. This situation is depicted in Figure 4.16. □

Similarly to the previous approach, we prove the correctness of the new ϵ-terminal event.

Lemma 4.15. *Let $\mathcal{N}$ be a 1-safe Petri net and $\mathcal{E}(\mathcal{N},\sigma) = \langle B, E, F, l\rangle$ a constrained unfolding of $\mathcal{N}$ guided by an observation σ. Moreover, let us consider a sequence of configurations $\vec{C} = (C_0, \ldots, C_n)$ such that:*

- $\forall C_i, C_{i+1}\,,\ C_i \subseteq C_{i+1}$,
- $C_n \subseteq E$,
- $C_0 = past\,(e)$, *where* $e = \epsilon$-companion (e'),
  $C_1 = past\left(e^0\right)$, *where* e^0 *is an* ϵ*-terminal event of type 4,*
  $C_i = C_{i-1} \oplus I\,(past\,(e') \setminus past\,(e))$, *where* e' *is an* ϵ*-terminal event from Definition 48,*

Then, the following statement is true:

$$\forall C_i, C_j \in \vec{C}\,,\ futures_{|E}\,(C_i) = futures_{|E}\,(C_j)\,.$$

Proof. Let us consider the graph G from the proof of Lemma 4.14. We will show that the procedure used to compute the value of $reps\,(G)$ is correct, *i.e.* it ensures that a set of futures for all the configurations in $\vec{C}$ is identical. In order to explain this, we prove the following property:

$$\forall C_i, C_j \in \vec{C}, \forall b_1 \in C_i, b_2 \in C_j\,,\ l\,(b_1) = l\,(b_2) \\ \Longrightarrow \text{obs}\,[past\,(b_1)] = \text{obs}\,[past\,(b_2)] \quad (4.1)$$

where $\text{obs}\,(E) = \{e \in E \mid \lambda\,[l\,(e)] \neq \epsilon\}$.

Such a property guarantees that all the possible extensions of C_i and C_j are identical in terms of constrained unfolding.

Let us recall the Algorithm 4.5: the graph G contains edges which represent two different relations.

- Edges with cost 1 represent causal links between the conditions inside a single segment of the considered ϵ-loop,

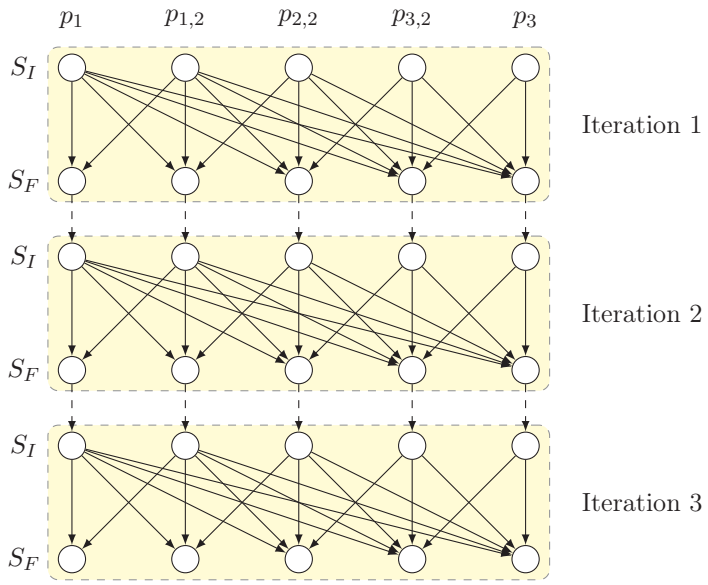


Figure 4.17: Illustration for the proof of Lemma 4.15.

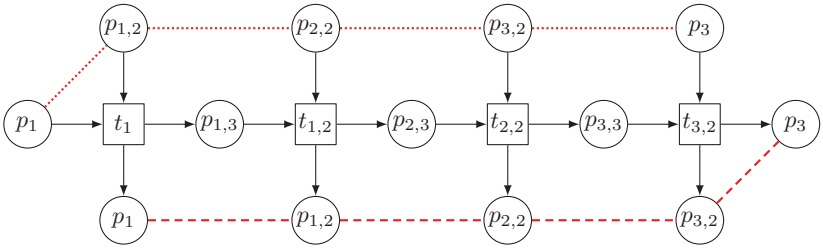


Figure 4.18: A fragment of the branching process in Figure 4.14 representing a segment of an ϵ-loop.

- Edges with cost 0 represent causal links between the conditions in the consecutive iterations of the ϵ-loop.

The procedure for the computation of $reps\,(G)$ takes all the pairs of the vertices and computes the minimum cost path for each pair. In other words, for each pair of the initial and the final conditions in the ϵ-loop, we get to know the number of iterations of the loop which is sufficient to causally connect the two conditions. In Figure 4.17, we present a schema explaining the main principle of the computation of the function $reps\,(G)$, where G is a graph in Figure 4.19. Now, when we take a pair with the greatest cost, we get the minimal number k of repetitions of the ϵ-loop such that Expression 4.1 is satisfied for all the configurations created as a result of lth iteration of the loop, where $l \geq k$. □

EXM **Example 28.** In Figure 4.18, we can see a fragment of the branching process

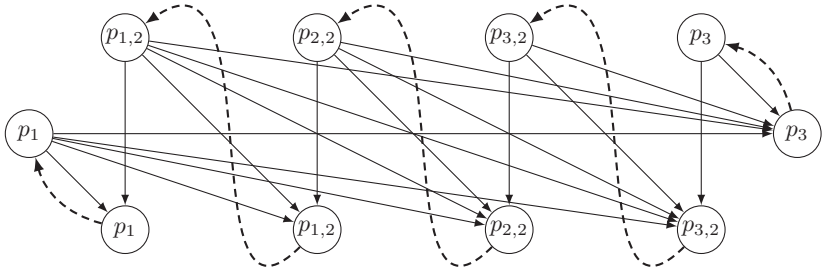


Figure 4.19: The weighted graph obtained from the segment in Figure 4.18.

representing a segment of an ϵ-loop. The dotted line groups conditions which belong to the state before execution of the loop. The dashed line groups conditions which belong to the state obtained after execution of the loop. Figure 4.19 represents the weighted graph obtained from the segment in Figure 4.18. The solid arcs have cost 1, the arcs drawn with dashed lines have cost 0. After the computation of the least cost paths between all pairs of vertices, we can note that the highest of all the costs is equal to 4. The path with this cost leads from the vertex p_3 to p_1. Indeed, in Figure 4.14, we can partly see that after four or more loops, the possible extensions after each iteration are the same. The result supervisor is presented in Figure 4.20.

4.5.4 Extraction of processes from constrained unfoldings

When using constrained unfoldings, in order to represent possible explanations of some observations, we may notice some unwanted effects. To understand the problem, let us consider a 1-safe Petri net in Figure 4.21a. In Figure 4.21b, we can observe the constrained unfolding built for observation $\sigma = \{a\}$. Note that we can not remove any of the events of the prefix without losing any explanation of the observation. However, let us note that such a prefix is also correct for observation $\{a, a\}$ and even $\{a, a, a\}$.

Before we look at the problem closer, we try to answer the question whether this problem also applies to certain special cases of 1-safe Petri net, namely a finite state machine and a free-labeled Petri net.

Lemma 4.16. *Every process of a constrained unfolding $\mathcal{E}(\mathcal{N}, \sigma)$ of an FSM $\mathcal{N}$ is a part of an explanation of the observation σ.*

Proof. If we take a transition t of $\mathcal{N}$ and all processes of the constrained unfolding such that t is maximal, we can note that t belongs only to one

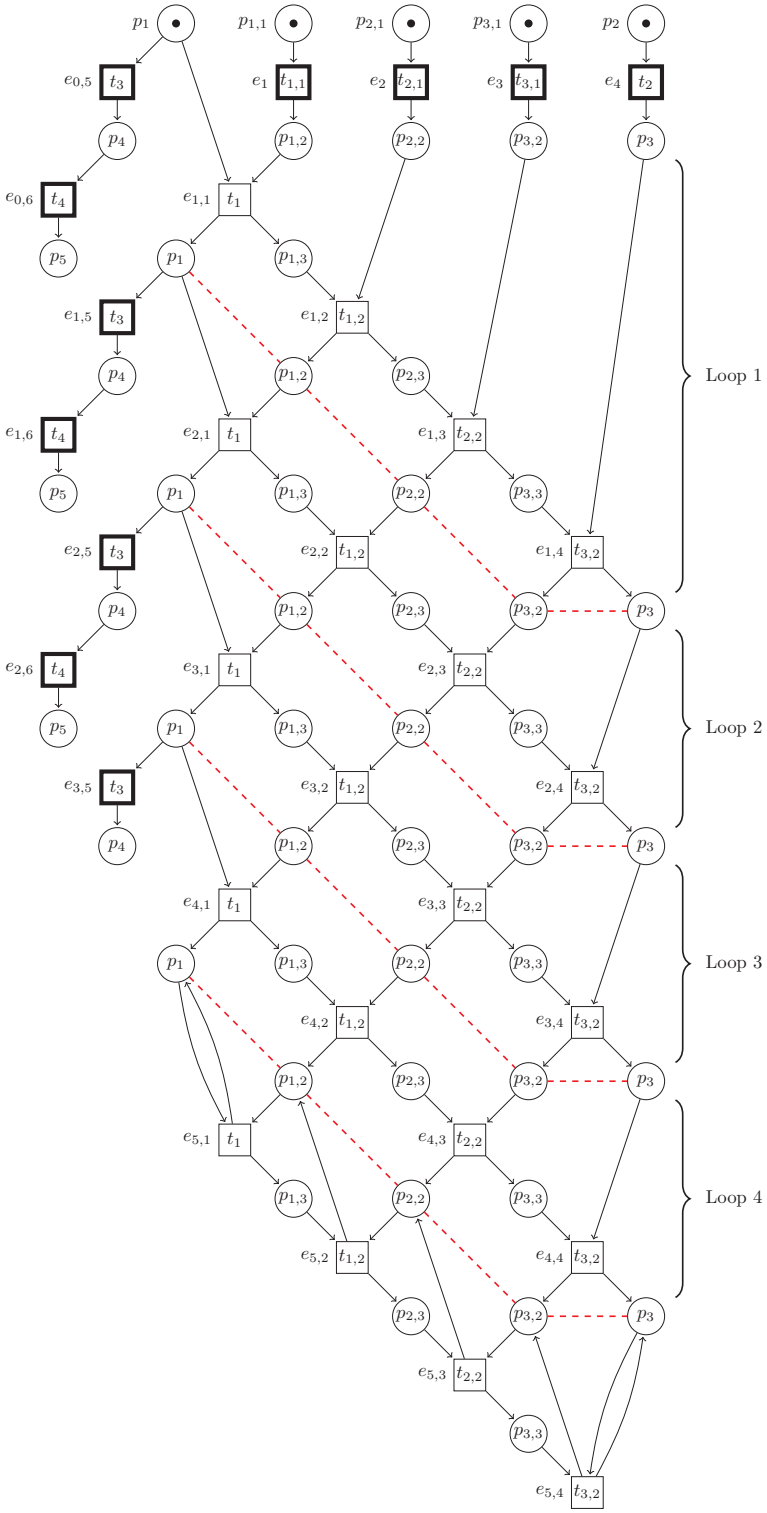


Figure 4.20: A correct supervisor of the net represented in Figure 4.13.

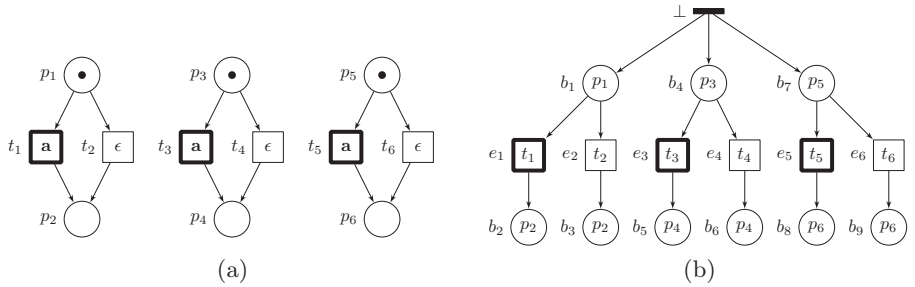


Figure 4.21: A 1-safe Petri net (a) and a constrained unfolding (b) valid for many different observations.

process. This is due to properties of finite state machines. We can not have parallel transitions which are not in conflict in FSMs. Thus, the transition t cannot belong to some unwanted explanation - explanation which exceeds the observation - because, when t was added to $\mathcal{E}(\mathcal{N}, \sigma)$, it was already checked that it satisfies the observation. □

Lemma 4.17. *Every process of a constrained unfolding $\mathcal{E}(\mathcal{N}, \sigma)$ of an FLPN $\mathcal{N}$ is a part of an explanation of the observation σ.*

Proof. To see that there are no unwanted explanations in FLPN, we can show that, for any co -set which is a cut of some configuration, the process does not exceed the observation. This can be easily observed, for example, by temporarily adding an unobservable event with a preset consisting of all conditions of the given coset. We can see that the Parikh vector of such event shows if the process exceeds the observation. From Lemma 4.5, we can also note that the vector can be calculated by simply taking the maximum of all Parikh vectors of all conditions. □

We already know that, when operating with constrained unfoldings of Petri nets, we can encounter erroneous processes. Now, we will introduce two notions which show some important aspects of constrained unfoldings and explanations which are coded in it. We can call these notions conflicting sets of events, *i.e.* sets of events which may normally create a correct process but are incorrect as explanations. Before we briefly describe the two types of conflicts, we want to emphasize that it is not sufficient to consider just pairs of events as it is in the case of standard conflict between events which cannot be in the same process. To see why, we can look at Figure 4.21a. Let us take an observation $\{a, a\}$. The prefix for the observation will be exactly like in Figure 4.21b. We can observe that, in a single explanation, any pair of observable events is actually possible. However, it is impossible to have 3 observable events (*i.e.* $\{a, a, a\}$) at once in an explanation.

The first type of conflict between events in a set C, expressed as $\#_1^\sigma C$, occurs when the events exceed the Parikh vector of the observation σ. Note that such a conflict can be captured at any stage of the construction of the prefix. Formally, such a conflict can be defined as follows:

$$\#_1^\sigma C \equiv \zeta(C) > \varpi(\sigma)$$

The second type of conflict is a little more complex, requiring the analysis of future events, between which we want to check the existence of a conflict. Namely, while the first type of conflict checks whether the observation vector has not been exceeded, the second type of conflict checks whether events may occur together in a process that fully satisfies the Parikh vector. Therefore, the test for the presence of this conflict requires that the analyzed structure contains all the possible explanations for this observation. This conflict can be expressed using the following formula:

$$\#_2^\sigma C \equiv \nexists \langle B, E, F, l\rangle \in processes(\mathcal{E}), C \subseteq E \wedge \zeta(E) = \varpi(\sigma)$$

The conflict of type 2 is more difficult to state than the conflict of type 1. Every time we want to verify if a set of events is in conflict of type 2, we have to analyze, not only the events of the set, but also its possible extensions.

EXM **Example 29.** In Figure 4.21b, we can see the prefix which is correct for observation $\sigma = \{a, a\}$. We mentioned earlier that such a prefix contains false processes that have events which can not be removed. Let us take such a process consisting of events $\perp, e_1, e_3$ and e_5. We can observe that the events are in conflict of type 1. Therefore, these events can not all occur in one explanation. Another process which is incorrect is a process of the events $\perp, e1, e4, e6$. In this case, there is a conflict of type 2. As we can note, the process comprising these events is not complying with the full observation vector. What is more, there are no perspective of extending this process within the given structure.

Remark that, we usually do not need any special knowledge about each possible process of a constrained unfolding. Actually, it is often sufficient to know that all the events belong to a *complete explanation*. Later, if necessary, all the explanations can be extracted by using the structure and definitions of the conflicts.

Chapter 5

Prototypes

In this chapter, we present several results and comments connected to the practical experiments we conducted during our studies. We describe below several main issues and problems we encountered during the implementation of our solutions.

5.1 Constrained unfoldings of networks of automata

As we described in Section 2.6.1 in case of networks of automata, we decided to apply a special event structure used to store and process constrained unfoldings. Since it is different from the solutions proposed for Petri nets, below we briefly present the main algorithms used to compute constrained unfoldings of network of automata. The algorithms were implemented and successfully used in case studies (see *e.g.* Section 3.3.2).

Before we describe the algorithm, we recall some notions which are applied in the context of unfoldings.

When talking about unfoldings and its construction, we have to consider two key issues: *search strategy* and *search scheme*. In simple words, search strategy tells us which event out of all the available events has the priority to be fired. For this purpose, for example we may choose the depth first search or the breadth first search based strategies. In turn search scheme indicates us two things: which event is a *terminal* one and can be extended, and which event is a successful one (one which we are searching for).

The algorithm below computes a prefix of a constrained unfolding network of automata for the given search scheme and search strategy. By a global transition we denote both local transition of automata and synchronizations. $\mathcal{T}$ stands for the set of all global transitions. For the convenience, we treat global transitions as sets of pairs (t, A), where t is a transition of an automaton A. T denotes a global transition. In the algorithm, we basically maintain three structures:

- the prefix $\mathcal{P}$ - a set of *complete* events (events for which there is a valid transition in the model) which is actually a directed acyclic graph used to keep the events in certain order,
- the collection of events E - a collection of events to be visited; it can be a list, a queue etc. Usually it depends on the search strategy.
- the collection of incomplete events $IncompleteE$ - a collection of events which are not complete in the sense that they consist of a strict subset of some global transition. When we take such two incomplete events e, f which are not in conflict, and when the sum of their sets of transitions is equal to some global transition, we can create a new valid event (complete or incomplete).

The function $pickNext()$ takes (according to the search strategy) and removes from E an event e which is to be visited. If the event is complete, it is added to the prefix and extended if possible. If not, it is merged with some other "pending" events (from $IncompleteE$) which are incomplete and belong to the same global transition. After each fusion of two events, we get a new complete or incomplete event. Moreover, we still keep the old incomplete events just in case we want to merge them with some other incomplete events which occur later during the construction of the prefix. $succ()$ is a function which computes all possible extensions of an event. In fact, the extensions form a set of events (complete or incomplete). $\uplus$ is used to sum local transitions of events and to create a new event.

In Algorithm 5.1 and Algorithm 5.2, there are some auxiliary functions which are defined as follows:

- IS-CONFLICT-FREE $(E) \stackrel{def}{=} \neg\exists f, g \in E, i \in \{1, \ldots, n\} . f \neq g \wedge \pi_i(f) = \pi_i(g)$
- IS-EXTENDABLE $(e) \stackrel{def}{=} \exists T \in \mathcal{T}, \tau(e) \subset T$
- IS-COMPATIBLE $(e, f) \stackrel{def}{=} \exists T \in \mathcal{T} . \tau(e) \subset T \wedge \tau(f) \subset T$
- IS-COMPLETE $(e) \stackrel{def}{=} \exists T \in \mathcal{T} . \tau(e) = T$
- IS-TERMINAL $(e) \stackrel{def}{=}$ event e is terminal according to the search scheme
- IS-CONFLICT-FREE $(\downarrow e) \iff$ isHistoryConflictFree $(e, \mathcal{P})$

In Algorithm 5.1, we do not consider observations which should be considered during the construction. This functionality can be built on the top of the presented algorithm (see Algorithm 5.3 for a part of the modified version with observations).

Algorithm 5.1 Pseudo-code of the prefix function for network of automata

```
1: function UNFOLD(𝒩, e_init) : 𝒫
2:     input 𝒩 : network of automata
3:           e_init : event                                  ▷ Initial event
4:     output 𝒫                                              ▷ Prefix
5:     var E : collection of events                          ▷ Events to visit
6:         IncompleteE : collection of events   ▷ Visited events but still
   active
7:         e, e′ : events

8:     procedure VISITANDEXTEND(g : event)
9:         𝒫 ← 𝒫 ∪ g                                        ▷ Visiting the new event
10:        if ¬IS-TERMINAL (g) then
11:            E ← E ⊕ succ (𝒩, g)
12:        end if
13:    end procedure

14:    begin
15:        E ← ∅
16:        IncompleteE ← ∅
17:        VISITANDEXTEND (e_init)
18:        while E ≠ ∅ do
19:            e ← pickNext (E)    ▷ Take an event according to the search
   strategy
20:            if IS-COMPLETE (e) then
21:                VISITANDEXTEND (e)
22:            end if
23:            if IS-EXTENDABLE (e) then
24:                for all f ∈ IncompleteE such that
   IS-COMPATIBLE(e, f)
25:                        ∧e′ ← e ⊎ f ∧ IS-CONFLICT-FREE (↓ e′) do
26:                    E ← E ⊕ e′
27:                end for
28:                IncompleteE ← IncompleteE ⊕ e
29:            end if
30:        end while
31:        return 𝒫
32:    end
33: end function
```

Algorithm 5.2 Procedure of searching for conflict in $\mathcal{P}$ in the history of e

```
 1: function isHistoryConflictFree(e, P) : boolean
 2:     input e : event
 3:           P : prefix
 4:     output boolean
 5:     var f, g : event
 6:         E, F : collection of events

 7:     function visitedSucc(h : event) : collection of events
 8:         G ← ∅
 9:         for all i ∈ succ (h, P) do
10:             if visited [i] then
11:                 G ← G ⊕ i
12:             end if
13:         end for
14:         return G
15:     end function

16:     begin
17:         for all  f ∈ P do
18:             visited [f] ← false
19:         end for
20:         E ← pred (e, P)
21:         while E ≠ ∅ do
22:             g ← pickNext (E)
23:             if visited [g] then
24:                 F ← visitedSucc (g, P)   ▷ Take direct successive events
    of g which were already visited
25:                 if ¬IS-CONFLICT-FREE (F ∪ {g}) then
26:                     return false                          ▷ A conflict detected
27:                 end if
28:             else                                          ▷ ¬visited (g)
29:                 visited [g] ← true
30:                 E ← E ⊕ pred (g, P)
31:             end if
32:         end while
33:         return true
34:     end
35: end function
```

Algorithm 5.3 A modified version of visitAndExtend procedure used in Algorithm 5.1.

```
1: procedure VISITANDEXTEND(g : event, σ : observation)
2:     ς(g) = Σ_{f∈↓g} χ_{λ(τ(f))}
3:     if ς(g) ≤ ϖ(σ) then
4:         P ← P ∪ g                                  ▷ Visiting the new event
5:         if ¬IS-TERMINAL(g) then
6:             E ← E ⊕ succ(N, g)
7:         end if
8:     end if
9: end procedure
```

The tool is written in Java with about 9000 lines of code. It has its own simple editor to edit models written in a special language used to define networks of timed automata.

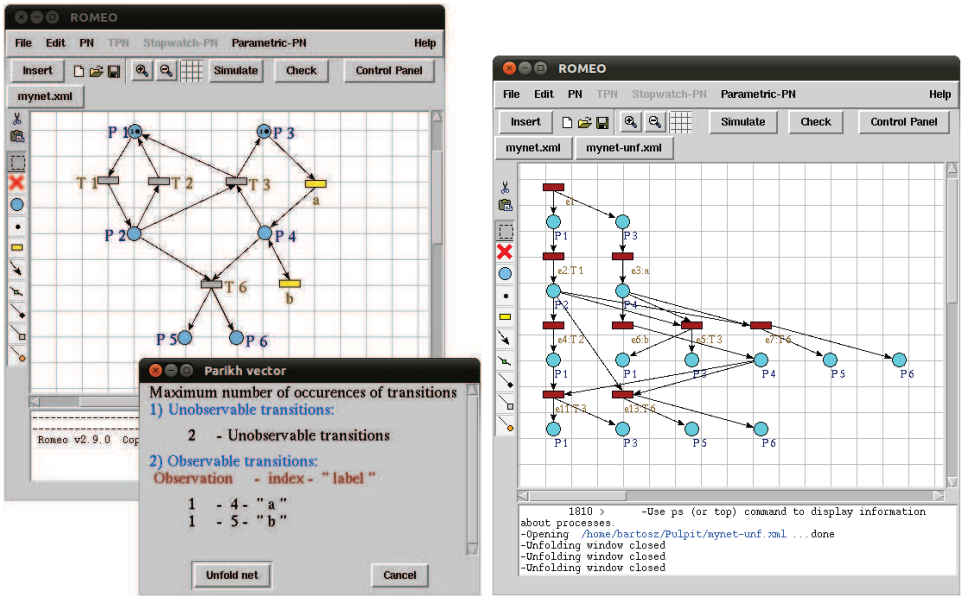
5.2 Constrained unfoldings of parametric time Petri nets

In Section 3.5.4, we mentioned an example of the alternating bit protocol which was implemented in the Roméo tool. The unfolding technique which was used in this case study is already well-described in [84]. Thus we only recall some main features connected to this tool and to the subject of constrained unfoldings.

In the current version of Roméo 2.9 we can define a parametric time Petri net and then we can compute a constrained unfolding on the basis of a given unstructured observation. In the tool we have the possibility to indicate unobservable transitions in the model. For the construction of constrained unfolding we can give a maximal number of unobservable loops for each event in the unfolding. We can also define a vector characterizing number of occurrences of events related to all the observable transitions. As a result we get a prefix of the constrained unfolding and constraints assigned to the events. Note that the problem of infinite unobservable loops is still an open issue in Roméo.

In the context of supervision, one of the advantages of this tool is certainly a possibility to use parameters in the time constraints of the model (the time constraints are handled with the Parma Polyhedra Library [10]). This enables us in many cases to discover possible valuations of parameters for a given behavior of the model (see for example Section 3.5.5).

On the other side, the unfolding technique, even in the case of constrained unfoldings, is still a challenge in the context of time and memory complexity. For more complex models with many unobservable transitions,



(a) (b)

Figure 5.1: A tool Roméo: a Petri net with two observable transitions and an observation (a) and an associated prefix of its constrained unfolding (b).

and unstructured observations, computation of constrained unfoldings might be in practice very difficult. For this reason, some additional assumptions which are possible in the context of supervision, could probably accelerate and simplify the construction of constrained unfoldings. This is however left for further studies.

5.3 Unobservable loops in Petri nets

During our studies we developed a tool which computes constrained prefixes and supervisors of Petri nets. The implementation is based on the approach described in Section 4.5.2.

The input of the program consists a Petri net defined in the file format of Roméo. In Figure 5.2a we can find an example of a 1-safe Petri net defined in Roméo. The Petri net has two observable transitions a and b. Then, for comparison in Figures 5.2b and 5.2c we may find two prefixes of a constrained unfolding based on observation $\{a, b\}$. Both prefixes are limited with the number of unobservable events in the history of each individual event. Thus in Figure 5.2b the limit is 4 and in Figure 5.2c it is 10. As we can observe the constrained unfolding is in general infinite. To solve this problem we can apply our tool and compute an associated constrained prefix. Such a prefix is presented in Figure 5.3b. To ease the comparison we give in Figure 5.3a a different presentation of the prefix from Figure 5.2b. Finally

using our tool we can compute a related supervisor (Figure 5.4) which is a folded version of the constrained prefix.

The current version of the software is limited to free-labeled Petri nets (see Definition 47).

Apart from the above mentioned features, the tool offers the following functionalities:

- unfolding of Petri nets with a maximum given depth. This feature helped us to discover two errors in Roméo. We found it also more efficient for Petri nets of bigger size.
- a comparison of two prefixes, *i.e.* the inclusion,
- dynamic addition of events. This feature is experimental and implements a more incremental way to construct constrained unfoldings. Thus in the beginning of the construction we do not have a whole observation. We assume that new events arrive during the construction of unfolding. At each moment of the construction we can take a snapshot of the constrained unfolding and continue later the procedure.

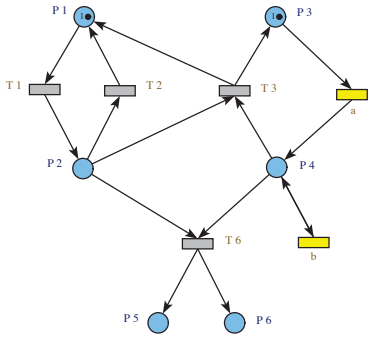
The tool was implemented in C++ and uses a Graphviz tool which is an open source graph visualization software.

5.4 Spinta

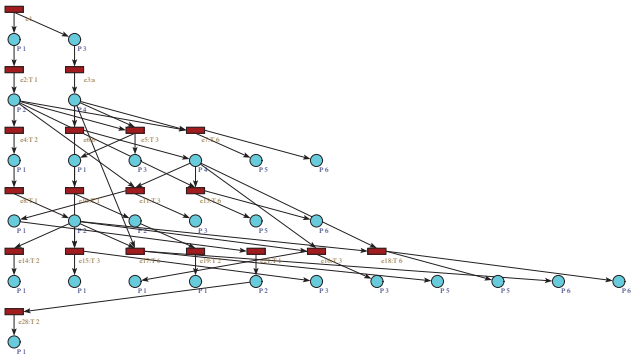
Spinta is an experimental tool which was created on the basis of our first tool used to construct constrained unfoldings of networks of timed automata (see Section 5.1). The current version of Spinta (version 0.1) performs computation of constrained prefixes of *networks of parametric automata with constraints represented by polyhedra* (NPA; see Section 2.2.5).

The main features of the program

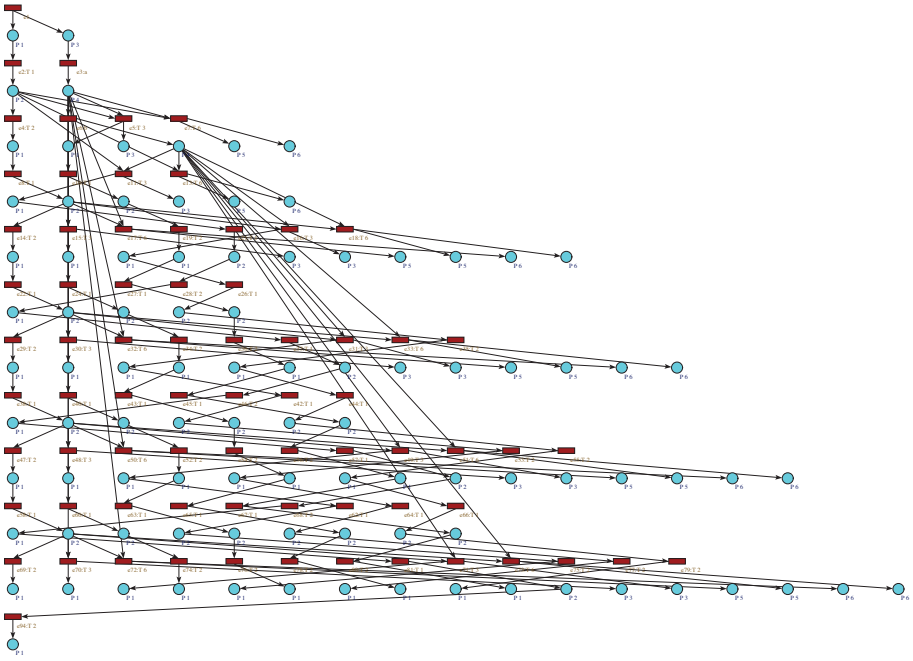
- The program includes a simple editor in which we can define *NPA* and verify its syntax.
- The program can compute constrained prefixes of *NPAs* on the basis of a given finite observation and additional parameters (see Section 5.4.1).
  - Extraction of explanations with their constraints.
  - Folding constraints, *i.e.* elimination of all variables except for parameters.
  - Computation of disjunction of all folded constraints.



(a)

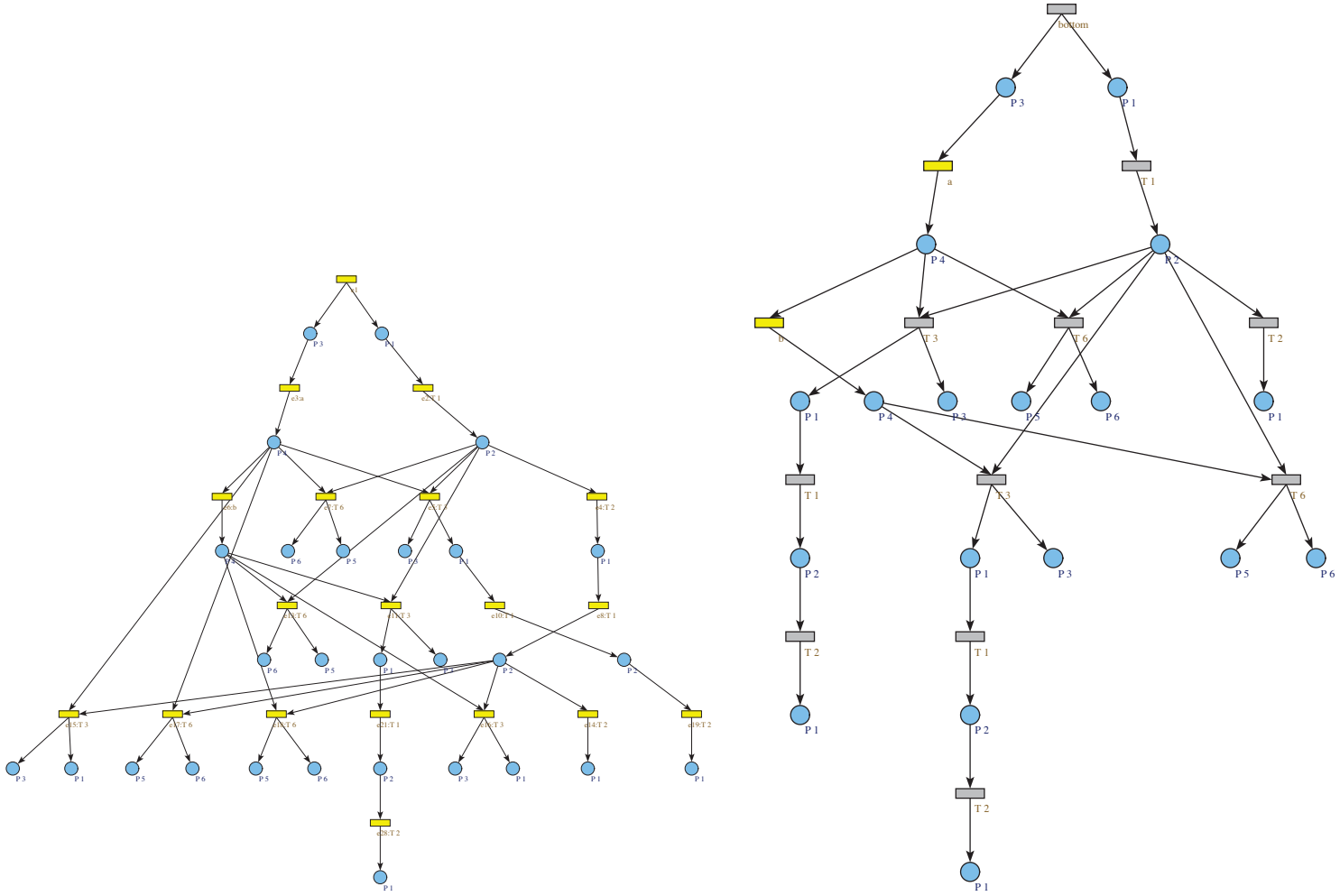


(b)



(c)

Figure 5.2: A 1-safe Petri net (a) and two prefixes of its constrained unfolding ((b) and (c)).



(a) (b)

Figure 5.3: A different form of the prefix in Figure 5.2b and a constrained prefix of the net in Figure 5.2a.

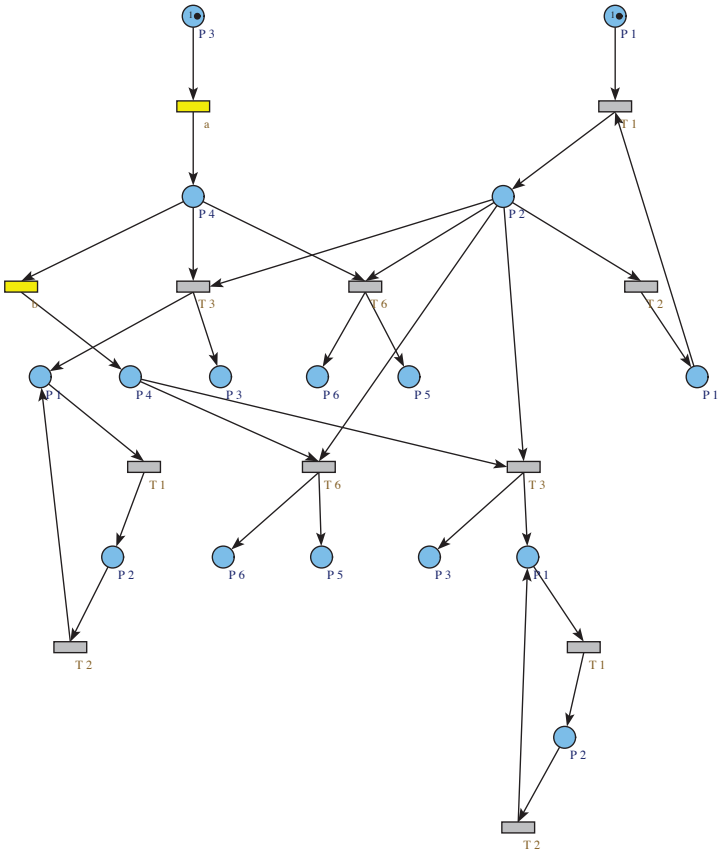


Figure 5.4: A supervisor of Petri net in Figure 5.2a.

– Producing a complete report from the computation process and a visualization of the prefix in *GraphViz* format (a file with extension "dot").

Below we describe the basic properties of the software. This section serves also as a short tutorial for Spinta.

5.4.1 Syntax of the model in Spinta

In order to define a network of automata in Spinta we have to use a dedicated language. The syntax of this language is described in details below.

Remark 5.1. We can export our model into *GraphViz* format (Model ▷ Export to .DOT). The model is saved in the file "_system.dot" in the same directory as the program.

Parameters and clocks

In the model, we can use two types of variables: clocks and parameters. There is no need to declare them before using in the model. However, we have to distinguish them in the code of the model. Namely, all parameters have to be prefixed with "_" (*e.g.* `_a, _param`). Consequently, clocks cannot start with "_" (*e.g.* `x, clock1`).

Constraints

In order to represent constraints, we can use a conjunction of linear constraints, *i.e.* $A\vec{x} \lhd \vec{b}$ in which $\lhd \in \{<, \leq\}$, A is a matrix of known coefficients, $\vec{x}$ is a vector of variables, $\vec{b}$ is a vector of known coefficients. In the Spinta code, we can express it by the following grammatical rules:

```
constraints : '(' constraint ('&&' constraint)* ')'
constraint : linear_expression
             ('<' | '<=' | '==' | '>=' | '>=')
             linear_expression
linear_expression : ('-' | '+')? (coefficient '*')* variable
                    (('-' | '+') (coefficient '*')* variable)*
```

All coefficients have to be integers.

For example, we can use the following constraint in Spinta:

```
( -2*x+_a-3*z <= 2*y-5+3*_b )
```

Package

A basic unit containing a model is called a *package*.

Table 5.1: The list of parameters.

Parameter name	Values	Default value
max_prefix_depth	$n \in \mathbb{N}$	3
observation	see Section 5.4.1	-
time_constraints	{true, false}	true
extract_explanations	{true, false}	false
filter_explanations	{true, false}	false
initial_constraints	see Section 5.4.1	-
folded_constraints	{true, false}	false
folded_constraints_together	{true, false}	false
only_max_explanations	{true, false}	true

```
package MyPackage;
/* Parameters */
/* Automata */
```

A package consists of two parts: a part with parameters and a part with automata.

Computation parameters

The parameters are used in the process of computation of a prefix. To specify the parameter *param* with a value *val,* we write:

```
@param val
```

The current version of Spinta supports the parameters in Table 5.1.

max_prefix_depth Describes the maximal depth of a prefix. It can be used together with observations.

observation Specifies a set of pairs (λ, n) where λ is a name of an observable transition or synchronization, and n is a number of events labeled by λ. There is a special symbol ? which describes unobservable events. It is used to bound the number of unobservable events in a prefix due to the problem of unobservable loops. By default, there is no observation and all events are valid. The value of this parameter should be the name of a file (in quotes) containing an observation defined by the following grammar rule $(\lambda_i, n_i)^*$.

time_constraints Indicates whether time constraints of a considered model are taken into consideration during the computation of a prefix.

extract_explanations If the value of this parameter is true, all the explanations of a given observation are extracted and are visible in a final report.

filter_explanations If the value of this parameter is true, there are only events which belong to explanations in the final diagram of the prefix.

initial_constraints Defines the initial constraints of a prefix. The constraints consist of a conjunction of strict or non-strict linear inequalities (*e.g.* `(2*x-3*y+z>=4*_a && z>_b)`). They are applied for the initial event in the prefix. Thus, before the constraints are used, values of all clocks are set to 0.

folded_constraints If the value of the parameter is true, all variables except parameters are eliminated from constraints of explanations.

folded_constraints_together If the value of the parameter is true, a disjunction of all folded constraints of all explanations is computed.

only_max_explanations If the value of the parameter is true, only the maximal (in the sense of inclusion) explanations are considered in the final result.

Automata

In each package, we can define a set of automata which constitute a network of automata. The definition of an automaton contains two types of definitions: locations and transitions.

```
automaton MyAutomaton {
        /* Locations */
        /* Transitions */
}
```

Locations The structure represents a single location of an automaton. The locations **do not** have to be declared before they are used in transitions.

```
modifier location name {
        constraints = ( /* constraints */ );
}
```

modifier In the current version of the program, there is only one type of modifier, *i.e.* `initial`. This type of modifier indicates the initial location of an automaton. Each automaton must have one initial location.

name A name of location. It is optional.

constraints An invariant assigned to the location. It is optional.

Transition The structure represents a single transition of an automaton.

```
modifier initial_location->final_location name {
        constraints = ( /* constraints */ );
        resets = { /* clocks */ };
        sync = { /* labels */ };
}
```

modifier In the current version of the program, there is only one type of modifier, *i.e.* `hidden`. This type of modifier indicates that the local transition is unobservable. It is optional.

name A name of a local transition which can be used in observations. It is optional.

constraints A guard assigned to the local transition. It is optional.

resets A set of clocks which are reset when the transition is executed. For example, `{x,y,z}`. It is optional.

sync A set of synchronizations considered by the local transition. For example, `{sync1, sync2, sync3}` where sync1, sync2, sync3 are labels of synchronizations. A name of synchronization which starts with character "`_`" means that the synchronization is *unobservable*. The parameter is optional.

Remark 5.2. In a model, we can mark both local transitions and synchronizations as unobservable. Unless a synchronization *sync* is unobservable, all events associated with this synchronization are visible in an observation as *sync*. Otherwise, if the synchronization is unobservable, labels of all its underlying observable transitions are put into the observation.

Comments

At any place in the code, we can insert comments. There are two types of comments presented below.

```
/*
 * A multiline comment
 */

// A single line comment
```

5.4.2 Computation of prefix

In order to compute a prefix of a model, we have to open it in Spinta. Then, from the menu, we choose Model ▷ Prefix ▷ Compute (the model has to be saved before the computation). The computation process is performed in a separate thread and can be terminated at any time (Model ▷ Prefix ▷ Terminate). The result consists of two files: a prefix in *GraphViz* format in the file "_prefix.dot" and a full report in "_report.txt".

5.4.3 Graphical user interface of Spinta

The GUI of Spinta consists of three parts (see Figure 5.5): the main part in which we can view and edit models (upper right panel), the panel with main properties of a chosen model (upper left panel) and the status window (the bottom panel).

Main panel

The main panel is an editor in which we can view and edit a model. To facilitate the programming of models, a pop-up menu is available from which we can chose all basic code structures and insert them directly into the model. The syntax of the model can be verified at any time (before we have to save our document) by choosing Model ▷ Verify Syntax from the menu .

Properties

After verification of the syntax or computation of a prefix, we can observe basic elements of the verified model in this window, *i.e.* automata, synchronizations, variables.

Status panel

In the status panel, we can observe all operations performed by Spinta. The contents of the status panel can be saved in an HTML file.

5.4.4 Final remarks

The program is written in Java (more than 11 000 lines of code). It uses the Parma Polyhedra Library to perform operations on polyhedra. The program generates files with diagrams which can be read in the program *GraphViz*.

The program package includes several examples.

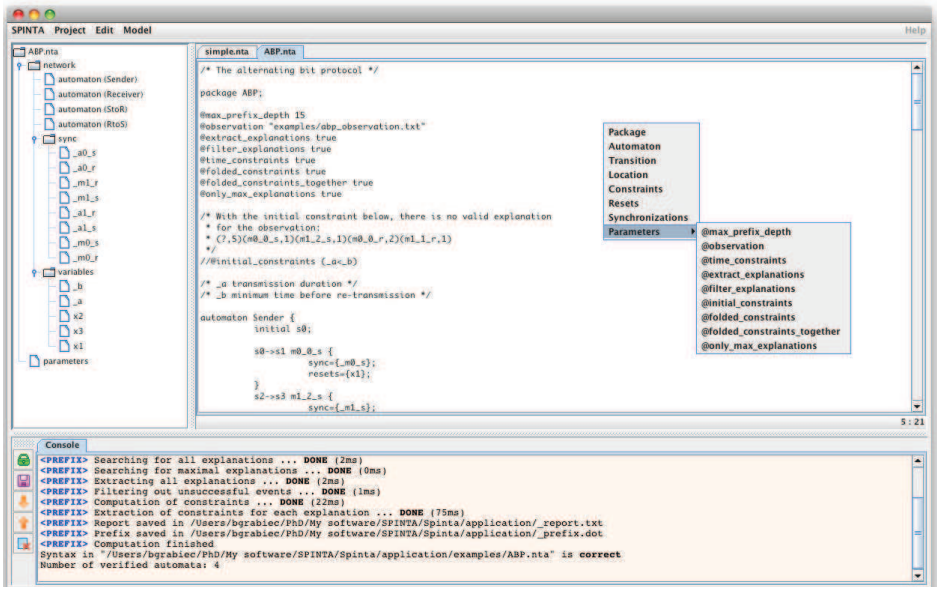


Figure 5.5: A screenshot of the program Spinta.

Chapter 6

Conclusion

6.1 Results

The problem of monitoring of distributed systems certainly is a subject that still remains open for many future researches. In our work, we presented a number of important issues that raise some important aspects of monitoring.

Distributed systems created today apply many advanced solutions. Nevertheless, despite the specific procedures used to create these systems, it often happens that they still require constant testing and monitoring in order to eliminate errors of various types. They are not necessarily errors which are complex and difficult to predict. There are many reasons for this, such as an excessive complexity of system or simply a lack of predictability of all the events that affect performance of the system. The reason we study in our work can be summarized as: a lack of or an inadequate formal verification of the model on which a concerned system is based, of course assuming that such a model exists, which is not often true. Of course, apart from the formal verification of the model, it is important that the model is properly implemented. Hence, the approach which we propose to solve the problem of monitoring of distributed systems tries to use formal methods and shows the possible advantages and disadvantages of such a formal approach in practice.

In this context, the desire to formalize all phases of software lifespan as much as possible seems to be reasonable. As already mentioned, the practice shows that the formal verification of the systems is often extremely difficult or even impossible to perform. And since for practical or theoretical reasons such problems can not be solved, this naturally leads to some solutions which try to solve it less formally or with less restrictions than assumed originally. Hence, there are numerous methods introduced in the context of software engineering.

In our work, we focused on monitoring as an essential element of verification of the correctness of distributed systems. We began our presentation by describing the main issues related to the monitoring of distributed systems.

Then, we briefly presented the notions which served us as a base for further discussion. In the solutions that we presented, we used two different models of formal models: time Petri nets and networks of timed automata. This allowed to show some specific features of both models.

As mentioned, usually it is not possible to fully verify a system before it is executed. However, sometimes it is possible to perform a limited verification of a system during its activity. It turns out that, when collecting incoming information about events in the system, we can often reduce enough the complexity of the problem we are interested in to be able to verify some properties and find explanations for certain behaviors of the system (*i.e.* scenarios of execution). For this purpose, we used the theory of unfoldings. Additionally, we presented this notion in the context of two different models. At the end, we studied the problem of unobservable events in distributed systems which are often ignored in practice, but that can be a valuable source of information about errors.

6.1.1 Unfoldings in supervision

We presented an original method using the model of networks of timed automata to produce timed explanations of a sequence of actions produced by a distributed system under surveillance [52]. For the purpose of our problem we develop the idea of constrained unfoldings which is derived from the theory of unfoldings. This way, we could obtain, not only information about causal relations between events of the underlying model, but also information about time at which the events occurred. Concerning the algorithmic complexity, we note that one of its main components is non-determinism caused by unobservable events. The more unobservable events in the partial observation, the greater the number of possible explanations of the model execution. This could also be observed during the tests which we performed to verify our solutions. In our work, we show that on the basis of the problem we presented, many possible applications can be considered such as the correlation of alarms and the detection of errors, monitoring behavior patterns to detect for example intrusions, surveillance of non-functional time properties, etc.

Following our research, we also apply a new technique for the unfolding of safe stopwatch parametric Petri nets ([53, 85, 84]) that allows a symbolic handling of both time and parameters. To the best of our knowledge, this is the first time that the parametric or stopwatch cases are addressed in the context of unfoldings. Moreover, when restricting to the subclass of safe time Petri nets, our technique compares well with the previous approach of [35]. It indeed provides a more compact unfolding, by eliminating the duplication of transitions, and also removes the need for read arcs in the unfolding. As a trade-off, the constraints associated with the firing times of events may seem slightly more complex.

Most of the solutions that we propose in the framework of the network timed automata-based surveillance and monitoring systems based on time Petri nets have been implemented and tested on various examples and some case studies, some of which are presented in this work. In addition, a solution based on the time Petri nets model is implemented as part of the earlier drop software tool for analyzing models based on Petri nets called Romeo. The current version of the Romeo tool 2.9.0 is available on the web-page [2]. It offers the possibility of computing symbolic unfoldings for safe time Petri nets with parameters. When guided by a sequence of actions, this feature allows the user to perform some diagnosis. The diagnosis consists of a finite prefix of the unfolding, presenting all the possible explanations of the input sequence. The explanations explicit the inferred causal relationships between the events of the model and also give the possible values for the parameters. We think that such an integrated method is a real added-value for the analysis of concurrent systems, and opens the door to deal with even more complex models like time Petri nets with stopwatches, or time Petri nets with more robust time semantics (*e.g.* with imperfect clocks).

6.1.2 Supervision for different models

In our work, we present the problem of monitoring from the perspective of two seemingly different models such as networks of timed automata and time Petri net. As mentioned in the introduction of our work, the models were developed relatively independently. More exact studies of relations between the models were conducted relatively recently. And there are still many open questions in this field. During our research, we tried to capture the characteristics that are common and unique to both formalisms, especially in the context of the problem of monitoring distributed systems. Differences between the two models are mainly due to differences in time constraints used in them.

For both networks of timed automata and time Petri nets, we have developed methods for monitoring distributed systems based on the aforementioned unfoldings. The methods also take into account the time constraints present in the models. Both approaches that we developed are also compared on a simple example of the alternating bit protocol [52, 84] partially prepared with software we implemented.

6.1.3 Unfoldings under partial observation

As we presented in the previous chapters, the prefix produced during the process of supervision can be theoretically infinite. This is due to the unobservable loops which can be executed in the system for an arbitrary number of times. That is why it is not possible to obtain all explanations for a given observation using the method introduced in [52, 53]. To tackle this problem

we show how to construct a constrained unfolding under partial observation, and then we explain why it is sufficient to only consider a finite part (a prefix) of it.

For the first time, we encountered the problem of unobservable loops during our research when discussing the monitoring distributed systems based on networks of timed automata (see article [52]). In the article, we propose a simple solution which is based on quite a strong assumption that the number of unobservable events in a single process of network of timed automata is bounded from above by a specified number. However, in practice, the assumption is quite a significant restriction. In addition, our test implementations only confirms that the number of unobservable events arising from the model of the system can dramatically affect memory and time complexity of the methods that we introduce to carry out the monitoring system. Consequently we decided to take a closer look at this problem.

In our further work, we address the particular question of unfolding Petri nets under partial observations. We propose a new method to deal with unobservable and infinite behaviors for constrained unfoldings of Petri nets. Throughout the work, we use the bottom-up approach to present our results. We start with the simple case of finite state machine to show the basic aspects of the problem. Then we consider a more generalized case of free-labeled 1-safe Petri nets in which each transition has a unique label assigned to it. Finally, we discuss the most general case of 1-safe Petri net. In the context of the solution, we discuss various aspects of our solutions and some related problems. We prove correctness and completeness of our translations concerning the assumptions mentioned before.

To explain the concept of the unfoldings under partial observations, we consider a system modeled by a Petri net which produces two types of events: observable and unobservable ones. By definition, only the execution of observable events can be visible outside the system. The information about observable events, called the observation, is collected by a certain mechanism, which we call the collector. In our work, we mainly consider most general case of the observation, which is a set of unordered events. One of the key issues in the problem is the fact that we do not want to lose the information about the unobservable events. We notice that, in many situations, it is crucial to have the information in order to be able to analyze behaviors that cannot be observed. It is not difficult to imagine a system which produces a lot of unobservable events which may cause some serious perturbations such as unwanted delays or strange valuations of parameters (for example for parametric time Petri nets). Finally, in the same way as in the case of supervision without unobservable events, the main goal of the supervision under partial observation method is to recreate, on the basis of the observable events, a structure consisting of all behaviors possible for the given observation. Thus, we get explanations similarly to the case when all events in the system are observable. However, in order to store all the ex-

planations, even the infinite ones, we show that the prefix produced during supervision is different. Moreover, to extract the explanations, we can convert the prefix into a Petri net whose unfolding is compatible with the initial observations gathered during the process of monitoring.

In order to verify our methods, we developed a tool which, on the basis of a given model and a set of observations, can construct the corresponding prefix. In order to compute the prefix, the program can apply different termination criteria. In the simplest case, the construction of the prefix can terminate when a certain maximal number of consecutive events are executed. In the more complex cases the program detects unobservable loops and stops extending the prefix when the partial observation is fulfilled and the corresponding processes can be extracted.

It should be stressed that the aim of our study is to keep information about unobservable events. However, there are solutions in which the information about unobservable events is insignificant and where only a fragment of the information, which influences the observable events, is kept. As mentioned in Section 4.1, the question of invisible loops still remains largely open. It cannot be simply circumvented by a model transformation since the elimination of unobservable transitions from 1-safe Petri nets is still an open and very difficult question [88].

6.2 Perspectives

Below, we present some possible developments of this work. The section is divided into two parts.

6.2.1 Short-term goals

One of the first targets for further development are issues related to **the theory of partial unfoldings**, *i.e.* unfoldings with possible infinite and unobservable behaviors.

Above all, it would be valuable to solve the problem of unobservable loops in the case of timed models such as time Petri nets. Also, the natural continuation of the work is the problem of unobservable loops in networks of timed automata. As we showed in Section 2.5, relatively recently, there has been some research done on the relations between the two models such as comparison of their expressivenesses. We think that it would be interesting to solve the problem directly for the network of timed automata to compare possible advantages of the models in this context. As the so-far experience shows, an exact translation between the two models is possible in many cases; but it may be a difficult task in practice (see Section 2.5.2).

Another interesting direction for the future work could be an application of trellises and an attempt to define a canonical form of the supervisors. The theory of trellises was introduced in [45]. They constitute a certain

alternative to the unfolding technique which we present in our work. First of all, they are more compact than the standard unfoldings we presented as the conflict relations are not unfolded in the trellises. Secondly, the trellises were found to have some interesting factorization properties which make them good candidates for distributed supervision. Namely, it appears that, for a system which can be expressed as a product of components, its trellis is the product of the trellises of these components. This topic was discussed in [68], where the authors moreover show a simple example of application to the problem of diagnosis in distributed systems.

It could be also interesting to extend our studies to **more complex models** such as parametric stopwatch Petri nets or non-safe models. The subject of stopwatch Petri nets was already discussed in one of our articles (see [85]) which seems to be an interesting target to study.

Finally, a detailed **algorithmic analysis** is still necessary. During our research on supervision of timed distributed systems, we analyzed several case studies. However, it would also be interesting to focus more on the aspect of performance of the algorithms used for supervision in order to make them more practical for real-life applications. Since the solutions we present are quite generic, we think that, in many cases, **stronger assumptions in the process of supervision** can still be made. For example, some observations may be equipped with extra information on causal relations about the events. Or maybe, in some cases, only a part of explanation is necessary without the need to consider all the components of the distributed system.

6.2.2 Long-term goals

So far, for both models that we present and which are used to solve the problem of monitoring, it is assumed that time flows equally across all components of the system and that its measurement is perfectly accurate. It is not difficult to observe that such a case is impossible in practice, especially when we deal with large distributed systems. It is assumed that the model of time, which is used in the systems, is perfect but does not always work in real-world applications. The difficulty of this issue is especially obvious when we look at the problem of synchronization of clocks in distributed systems, and protocols created to solve it such as Network Time Protocol (NTP – for more information, see [74, 1]). Until now, many different solutions have been proposed to solve **the problem of reliability of the measurement of time in timed systems**, for both networks of timed automata and time Petri nets. There are many different approaches that try to solve this problem. One of them is based on systems which use standard models (with assumption of perfect clocks), without changing their semantics. This type of approach tries to avoid the problem of precision of clocks by using structural properties of the model (see [4]). Another type of solutions applies

systems based on models with a specially adapted semantics which takes into account different clock rates, offsets, and other significant properties of the clocks in distributed systems (for example [39, 55, 89, 40]). The selection of an appropriate model for designing distributed systems is not a simple task, especially when we take into account imperfections of clocks in such systems, and the fact that the problem of monitoring itself is relatively complex computationally. Moreover, additional features of the models with modified semantics raise a lot of new questions. The properties that were so far verifiable in the original model can not be checked for its new enhanced version.

The previous problem also implies another important issue: **implementability**. At the stage of designing a system, it is almost impossible to predict all the situations which may occur during its activity. Thus, finding a model that would perfectly fit the problem is very hard. Hence, the idea of using parameters in the models. An interesting direction of research that would expand our work may be automatic adaptation of some parameters (*e.g.* in time constraints) of a distributed system, depending on the environment in which the system operates.

From the perspective of implementation of the proposed solutions, an important issue certainly is the complexity of the algorithms. This is a problem which still needs theoretical analysis and a large number of tests. As mentioned earlier, the technique of unfoldings is interesting especially for two reasons: the information that can be stored using unfoldings and the complexity of memory that is needed to store this information. Unfortunately, it is usually associated with the large computational cost. In the case of monitoring, the size of the unfoldings is limited by various additional factors such as: observations collected by the monitoring module, and time constraints present in the model which eliminate some scenarios of the system's behavior.

Another direction that could be an interesting continuation of this work is to extend the functionality of monitoring system with **supervisory control**. In this case, the system would not only be monitored to determine whether there are some particular events or behaviors, but also there would exist a possibility of an automatic response to a number of situations. The idea of an automatic control of systems is not new and is a natural extension of the problem of supervision.

Bibliography

[1] NTP: The Network Time Protocol. http://ntp.org, 2011. 178

[2] Roméo: A tool for Time Petri Nets analysis. http://romeo.rts-software.org, 2011. 17, 35, 111, 175

[3] P. A. Abdulla, S. P. Iyer, and A. Nylen. Unfoldings of unbounded Petri nets. In *Proceedings of CAV*, volume 1855 of *LNCS*, pages 495–507. Springer, 2000. 75

[4] K. Altisen and S. Tripakis. Implementation of timed automata: An issue of semantics or modeling? *Formal Modeling and Analysis of Timed Systems*, pages 273–288, 2005. 178

[5] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *5th IEEE Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, Pennsylvania, Jun 1990. IEEE Computer Society Press. 57

[6] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 45, 47, 57

[7] R. Alur, L. Fix, and T. A. Henzinger. A determinizable class of timed automata. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 1–13, London, UK, 1994. Springer-Verlag. 59

[8] T. Aura and J. Lilius. A causal semantics for time Petri nets. *Theoretical Computer Science*, 243(2):409–447, 2000. 100

[9] R. Bagnara, P. M. Hill, and E. Zaffanella. Not necessarily closed convex polyhedra and the double description method. *Formal Aspects of Computing*, 17:222–257, 2005. 10.1007/s00165-005-0061-1. 50

[10] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008. 104, 161

[11] S. Balaguer, Th. Chatain, and S. Haar. A concurrency-preserving translation from time Petri nets to networks of timed automata. In *Proceedings of the 17th International Symposium on Temporal Representation and Reasoning (TIME'10)*, Paris, France, Sept. 2010. IEEE Computer Society Press. To appear. 62, 64

[12] P. Baldan, N. Busi, A. Corradini, and G. M. Pinna. Functorial concurrent semantics for petri nets with read and inhibitor arcs. In *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 442–457. Springer, 2000. 75

[13] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957. 57

[14] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, pages 87–124, 2004. 57

[15] A. Benveniste, E. Fabre, S. Haar, and C. Jard. Diagnosis of asynchronous discrete-event systems: a net unfolding approach. *IEEE Transactions on Automatic Control*, 48(5):714–727, 2003. 15, 17, 33, 35

[16] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. Roux. Comparison of the expressiveness of timed automata and time Petri nets. *Formal Modeling and Analysis of Timed Systems*, pages 211–225, 2005. 61, 63

[17] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux. When are timed automata weakly timed bisimilar to time Petri nets? *Theoretical Computer Science*, 403(2-3):202–220, 2008. 64

[18] B. Bérard, P. Gastin, and A. Petit. On the power of non-observable actions in timed automata. *STACS 96*, pages 255–268, 1996. 16, 34

[19] B. Bérard, A. Petit, V. Diekert, and P. Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36(2):145–182, 1998. 16, 34, 58

[20] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE trans. on Soft. Eng.*, 17(3):259–273, 1991. 55, 59, 112

[21] B. Berthomieu, D. Lime, O. H. Roux, and F. Vernadat. Reachability problems and abstract state spaces for time Petri nets with stopwatches. *Journal of Discrete Event Dynamic Systems - Theory and Applications (DEDS)*, 17(2):133–158, 2007. 59, 75, 112

[22] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. In *Information Processing: proceedings of the IFIP congress 1983*, volume 9 of *IFIP congress series*, pages 41–46. Elsevier Science Publishers, Amsterdam, 1983. 59

[23] E. Best and H. Wimmel. Reducing k-safe Petri nets to pomset-equivalent 1-safe Petri nets. In *Proceedings of the 21st international conference on Application and theory of petri nets*, pages 63–82. Springer-Verlag, 2000. 53

[24] P. Bouyer and F. Chevalier. On conciseness of extensions of timed automata. *Journal of Automata, Languages and Combinatorics*, 10(4):393, 2005. 58

[25] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Updatable timed automata. *Theoretical Computer Science*, 321(2-3):291–345, 2004. 58

[26] P. Bouyer, S. Haddad, and P. Reynier. Timed unfoldings for networks of timed automata. *Automated Technology for Verification and Analysis*, pages 292–306, 2006. 15, 34, 69

[27] P. Bouyer, S. Haddad, and P. Reynier. Undecidability results for timed automata with silent transitions. *Fundamenta Informaticae*, 92(1):1–25, 2009. 16, 34

[28] C. G. Cassandras and S. Lafortune. *Introduction to discrete event systems, Second Edition.* Springer, 2008. 16, 34, 75

[29] F. Cassez, T. Chatain, and C. Jard. Symbolic unfoldings for networks of timed automata. In *Proc. of the 4th International Symposium on Automated Technology for Verification and Analysis, ATVA 2006, number 4218 in Lecture Notes in Computer Science*, pages 307–321. Springer–Verlag, 2006. 15, 34, 69, 84

[30] F. Cassez and O. Roux. From time petri nets to timed automata. In *Fourth International Workshop on Automated Verification of Critical Systems (AVoCS'04)*, 2004. 63

[31] F. Cassez and O. Roux. Structural translation from time Petri nets to timed automata. *Journal of Systems and Software*, 79(10):1456–1468, 2006. 63

[32] T. Chatain. *Symbolic unfoldings of high-level Petri nets and application to supervision of distributed systems.* PhD thesis, University of Rennes 1, November 2006. 15, 34

[33] T. Chatain and J. Claude. Sémantique concurrente symbolique des réseaux de Petri saufs et dépliages finis des réseaux temporels. *Proceedings of NOTERE, Tozeur, Tunisia. IEEE Computer Society Press, Los Alamitos*, May-June 2010. 65, 75

[34] T. Chatain and C. Jard. Time supervision of concurrent systems using symbolic unfoldings of time Petri nets. *Formal Modeling and Analysis of Timed Systems*, pages 196–210, 2005. 15, 34, 69, 76

[35] T. Chatain and C. Jard. Complete finite prefixes of symbolic unfoldings of safe time Petri nets. In *Proceedings of ICATPN*, volume 4024 of *LNCS*, pages 125–145. Springer, 2006. 75, 76, 98, 111, 174

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, 2nd revised edition edition, September 2001. 149

[37] D. Corona, A. Giua, and C. Seatzu. Marking estimation of Petri nets with silent transitions. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 1, pages 966–971. IEEE, 2005. 16, 34

[38] D. D'Aprile, S. Donatelli, A. Sangnier, and J. Sproston. From time Petri nets to timed automata: an untimed approach. In *TACAS'07: Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, pages 216–230, Berlin, Heidelberg, 2007. Springer-Verlag. 64

[39] C. Daws and P. Kordy. Symbolic robustness analysis of timed automata. *Formal Modeling and Analysis of Timed Systems*, pages 143–155, 2006. 179

[40] M. De Wulf, L. Doyen, N. Markey, and J. Raskin. Robust safety of timed automata. *Formal Methods in System Design*, 33(1):45–84, 2008. 179

[41] V. Diekert, P. Gastin, and A. Petit. Removing ε-transitions in timed automata. In *STACS 97*, pages 583–594. Springer, 1997. 16, 34

[42] J. Esparza. Model checking using net unfoldings. In *TAPSOFT '93: Selected papers of the colloquium on Formal approaches of software engineering*, pages 151–195, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V. 14, 33, 75, 115

[43] J. Esparza and K. Heljanko. A new unfolding approach to ltl model checking. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 475–486, London, UK, 2000. Springer-Verlag. 16, 34

[44] J. Esparza and K. Heljanko. *Unfoldings: A partial-order approach to model checking*. Springer Publishing Company, Incorporated, 2008. 14, 33, 75, 79, 115

[45] E. Fabre. Trellis processes: a compact representation for runs of concurrent systems. *Discrete Event Dynamic Systems*, 17(3):267–306, 2007. 177

[46] E. Fabre, A. Benveniste, S. Haar, and C. Jard. Distributed monitoring of concurrent and asynchronous systems. *Journal of Discrete Event Systems, special issue*, pages 33–84, 5 2005. 15, 33, 76, 113

[47] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66, 1988. 70

[48] C. Fidge. Logical time in distributed computing systems. *COMPUTER,*, pages 28–33, 1991. 9, 29

[49] I. Foster. What is the grid? a three point checklist. *GRID today*, 1(6):32–36, 2002. 2, 22

[50] G. Gardey, D. Lime, M. Magnin, and O. Roux. Romeo: A tool for analyzing time Petri nets. In *Computer Aided Verification*, pages 418–423. Springer, 2005. 17, 35, 59, 63

[51] A. Giua. Petri net state estimators based on event observation. In *Proceedings of the IEEE ICDC*, pages 4086–4091, 1997. 76

[52] B. Grabiec and C. Jard. Unfolding of networks of automata and their application in supervision. In A. Obaïd, editor, *9th Annual International Conference on New Technologies of Distributed Systems (NOTERE 2009)*, Montreal, Canada, July 2009. 15, 34, 76, 104, 114, 174, 175, 176

[53] B. Grabiec, L.-M. Traonouez, C. Jard, D. Lime, and O. H. Roux. Diagnosis using unfoldings of parametric time Petri nets. In K. Chatterjee and T. A. Henzinger, editors, *8th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2010)*, Lecture Notes in Computer Science, Vienna, Austria, Sept. 2010. Springer. To appear. 76, 174, 175

[54] B. Grahlmann. The PEP tool. In *Computer Aided Verification*, pages 440–443. Springer, 1997. 17, 35

[55] V. Gupta, T. Henzinger, and R. Jagadeesan. Robust timed automata. In *Hybrid and Real-Time Systems*, pages 331–345. Springer, 1997. 179

[56] S. Haar, L. Kaiser, F. Simonot-Lion, and J. Toussaint. Equivalence of timed state machines and safe TPN. In *Discrete Event Systems, 2002. Proceedings. Sixth International Workshop on*, pages 119–124. IEEE, 2003. 63

[57] T. Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998. 76

[58] J. Jessen, J. Rasmussen, K. Larsen, and A. David. Guided controller synthesis for climate controller using Uppaal-Tiga. In *Proceedings of the 19th International Conference on Formal Modeling and Analysis of Timed Systems*, volume 4763 of *LNCS*, pages 227–240. Springer, 2007. 104

[59] N. D. Jones, L. H. Landweber, and Y. E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science 4*, pages 277–299, 1977. 59

[60] V. Khomenko and M. Koutny. Branching processes of high-level Petri nets. In *Proceedings of TACAS*, volume 2619 of *LNCS*, pages 458–472. Springer, 2003. 75

[61] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state encoding conflicts in STG unfoldings using SAT. *Fundamenta Informaticae*, 62(2):221–241, 2004. 17, 35

[62] V. Khomenko, M. Koutny, and A. Yakovlev. Logic synthesis for asynchronous circuits based on STG unfoldings and incremental SAT. *Fundamenta Informaticae*, 70(1):49–73, 2006. 17, 35

[63] K. Larsen and W. Yi. Time abstracted bisimulation: Implicit specifications and decidability. In *Mathematical Foundations of Programming Semantics*, pages 160–176. Springer, 1994. 57

[64] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997. `http://www.uppaal.com`. 57

[65] S. Lasota and I. Walukiewicz. Alternating timed automata. *ACM Transactions on Computational Logic (TOCL)*, 9(2):1–27, 2008. 59

[66] D. Lime and O. Roux. State class timed automaton of a time Petri net. In *Petri Nets and Performance Models, 2003. Proceedings. 10th International Workshop on*, pages 124–133. IEEE, 2003. 63

[67] D. Lime and O. Roux. Model checking of time Petri nets using the state class timed automaton. *Discrete Event Dynamic Systems*, 16(2):179–205, 2006. 63

[68] A. Madalinski and E. Fabre. Modular construction of finite and complete prefixes. In *Workshop Organisation*, page 69, 2009. 178

[69] F. Mattem. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989. 9, 29, 70

[70] K. L. McMillan. Using unfolding to avoid the state space explosion problem in the verification of asynchronous circuits. In *Proceedings of CAV*, volume 663 of *LNCS*, pages 164–177. Springer, 1992. 14, 33, 75

[71] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 164–177, London, UK, 1993. Springer-Verlag. 115

[72] P. M. Merlin. *A study of the recoverability of computing systems*. PhD thesis, Dep. of Information and Computer Science, University of California, Irvine, CA, 1974. 53, 55, 112

[73] R. Meyer, V. Khomenko, and T. Strazny. A practical approach to verification of mobile systems using net unfoldings. *Fundamenta Informaticae*, 94(3):439–471, 2009. 17, 35

[74] D. L. Mills. *Computer network time synchronization: The Network Time Protocol*. CRC Press, Inc., Boca Raton, FL, USA, 2006. 3, 23, 178

[75] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In K. Larsen and A. Skou, editors, *Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 376–398. Springer Berlin / Heidelberg, 1992. 41

[76] J. Ouaknine and J. Worrell. On the language inclusion problem for timed automata: Closing a decidability gap. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 54–63. IEEE, 2004. 59

[77] J. L. Peterson. *Petri net theory and the modeling of systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981. 123

[78] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962. 52

[79] C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1974. Project MAC Report MAC-TR-120. 53

[80] J. Raskin. An introduction to hybrid automata. *Handbook of networked and embedded control systems*, pages 491–517, 2005. 58

[81] C. Schröter, S. Schwoon, and J. Esparza. The model-checking kit. In *Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003. Proceedings*, pages 1021–1022. Springer, 2003. 17, 35

[82] J. Srba. Comparing the expressiveness of timed automata and timed extensions of Petri nets. In *Proceedings of the 6th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'08)*, volume 5215 of *LNCS*, pages 15–32. Springer-Verlag, 2008. 64

[83] C. Stehno. Real-time systems design with PEP. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 476–480, 2002. 17, 35

[84] L. M. Traonouez. *Vérification et dépliages de réseaux de Petri temporels paramétrés*. PhD thesis, Ecole centrale de Nantes, available at http://www.dsi.unifi.it/~traonove/publications/these.pdf, 2009. 15, 34, 98, 101, 103, 161, 174, 175

[85] L.-M. Traonouez, B. Grabiec, C. Jard, D. Lime, and O. H. Roux. Symbolic unfolding of parametric stopwatch Petri nets. In A. Bouajjani and W.-N. Chin, editors, *8th International Symposium on Automated Technology for Verification and Analysis (ATVA 2010)*, Lecture Notes in Computer Science, Singapore, Sept. 2010. Springer. To appear. 15, 34, 76, 97, 98, 112, 174, 178

[86] L.-M. Traonouez, D. Lime, and O. H. Roux. Parametric model-checking of stopwatch Petri nets. *Journal of Universal Computer Science*, 15(17):3273–3304, Dec. 2009. 59, 75, 114

[87] T. Ushio, I. Onishi, and K. Okuda. Fault detection based on Petri net models with faulty behaviors. In *Proceedings of the IEEE Int. Conf. on Systems, Man, and Cybernetics*, pages 113–118, 1998. 76

[88] H. Wimmel. Eliminating internal behaviour in Petri nets. *Applications and Theory of Petri Nets 2004*, pages 411–425, 2004. 16, 35, 114, 177

[89] M. Wulf, L. Doyen, and J. Raskin. Almost ASAP semantics: From timed models to timed implementations. *Formal Aspects of Computing*, 17(3):319–341, 2005. 179

List of Figures

Index

Résumé

Ce travail est consacré à la problématique du suivi des systèmes répartis temps réel. Plus précisément, il se concentre sur les aspects formels de la supervision basée sur des modèles ainsi que sur les problèmes qui lui sont liés.

Dans la première partie du travail, nous présentons les propriétés de base de deux modèles formels bien connus utilisés pour la modélisation de systèmes répartis : les réseaux d'automates temporisés et les réseaux de Petri temporels. Nous montrons que le comportement de ces modèles peut être représenté par les procédés dits de branchement. Nous introduisons également les éléments conceptuels clés du système de surveillance.

La deuxième partie du travail est consacrée à la question des dépliages avec contraintes qui permettent le suivi des relations causales entre les événements dans un système réparti. Ce type de structure peut reproduire des processus sur la base d'un ensemble totalement non-ordonné d'évènements. Dans notre travail, nous soulevons les problèmes des contraintes de temps et de leurs paramétrages. Les méthodes proposées sont illustrées par des études de cas.

La troisième partie du travail traite de la problématique des boucles inobservables qui peuvent résulter de comportements cycliques inobservables des systèmes considérés. Ce type de comportement conduit à un nombre infini d'événements dans les dépliages avec contraintes.

La quatrième et dernière partie du travail est consacrée à l'implémentation des méthodes décrites précédemment.

Mots clés :

Supervision, surveillance, systèmes répartis temps réel, réseaux de Petri temporels, réseaux d'automates temporisés, processus de branchement, dépliages avec contraintes, contraintes temporelles paramétrisées, boucles inobservables, observations partielles.

Abstract

This work is devoted to the issue of monitoring of distributed real-time systems. In particular, it focuses on formal aspects of model-based supervision and problems which are related to it.

In its first part, we present the basic properties of two well-known formal models used to model distributed systems: networks of timed automata and time Petri nets. We show that the behavior of these models can be represented with so-called branching processes. We also introduce the key conceptual elements of the supervisory system.

The second part of the work is dedicated to the issue of constrained unfoldings which enable us to track causal relationships between events in a distributed system. This type of structure can be used to reproduce processes of the system on the basis of a completely unordered set of previously observed events. Moreover, we show that time constraints imposed on a system and observations submitted to the supervisory system can significantly affect a course of events in the system. We also raise the issue of parameters in time constraints. The proposed methods are illustrated with case studies.

The third part of the work deals with the issue of unobservable cyclical behaviors in distributed systems. This type of behaviors leads to an infinite number of events in constrained unfoldings. We explain how we can obtain a finite structure that stores information about all observed events in the system, even if this involves processes that are infinite due to such unobservable loops.

The fourth and final part of the work is dedicated to implementation issues of the previously described methods.

Keywords:

Supervision, monitoring, distributed real-time systems, time Petri nets, networks of timed automata, branching processes, constrained unfoldings, time constraints with parameters, unobservable loops, partial observations.

N° d'ordre :

École normale supérieure de Cachan - Antenne de Bretagne

Campus de Ker Lann - Avenue Robert Schuman - 35170 BRUZ

Tél : +33(0)2 99 05 93 00 - Fax : +33(0)2 99 05 93 29